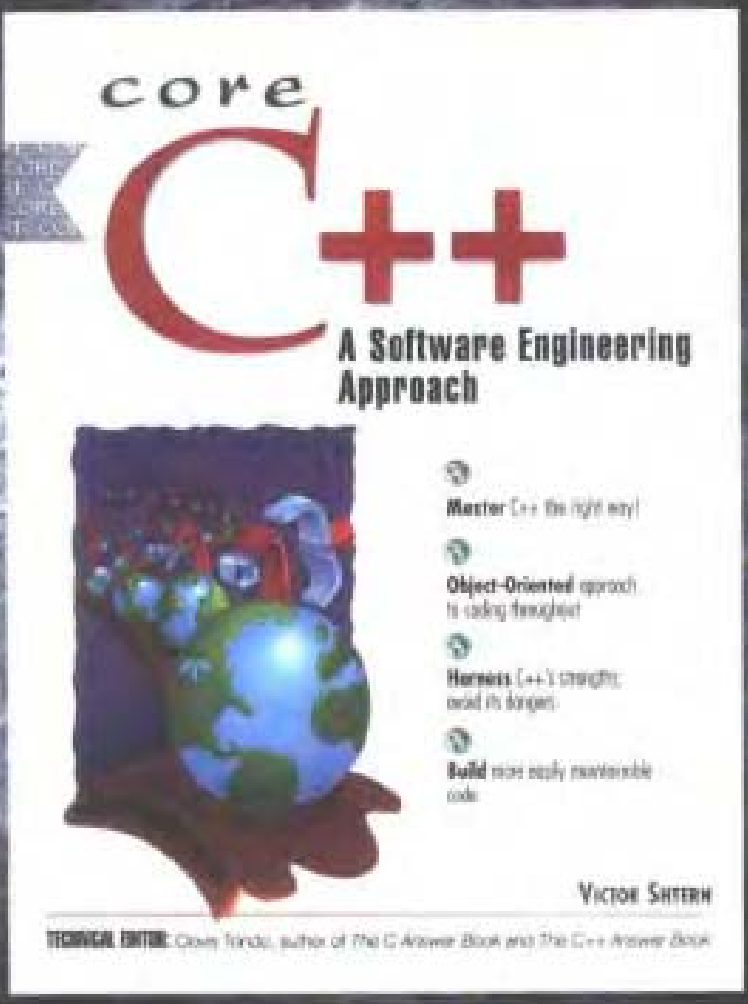
计 算 机 科 学 丛 书

C++精髓

软件工程方法

(美) Victor Shtern 著 李师贤 张珞玲 刘斌 郑红 等译
波士顿大学



Core C++

A Software Engineering Approach

机械工业出版社
China Machine Press
PH PTR

Shtern首先发现了如何编写可维护软件的技术。对于任何层次的程序员而言，本书都是富有直知灼见而又易于阅读的。它提出了许多有创含实际的建议和有用的例子

Daniel F. Costello
通用电气马泰利医疗系统公司高级软件工程师

无论对于初学者还是有经验的C语言的程序员来说，本书都有助于实现他们全面地理解C++语言以及基本的面向对象概念的愿望

Steve Glass
摩托罗拉公司高级软件工程师

本书以正确的方法对具有使用任何一种语言经验的开发人员讲授C++：即在C++程序设计中应用最好的软件工程实践和方法。即使开发人员已经使用过C++，这本内容广泛的书仍然可以教给读者如何创建更加健壮、更易于维护和修改以及更有价值的代码。

Shtern的这本书在讲授语言之前讲解了面向对象的原理，有助于运用面向对象开发方法的全面功能创建高级软件。本书帮助读者从软件工程师的角度掌握ANSI/ISO C++的每一种重要特性：类、方法、const修饰符、动态内存管理、类复合、继承、多态、I/O等。

如果希望创建优秀的C++软件，就应该使用当今最好的软件工程实践方法去设计、思考和进行程序开发。

本书重点内容：

- 软件工程原理在C++程序设计中的应用
- 重点强调编写将来容易维护和修改的代码
- 在教授语言之前实际地理解面向对象原理
- 深入分析最新ANSI/ISO C++的特点
- 几百个实际而中肯的代码示例

作者简介

Victor Shtern 是波士顿大学大都会学院的教授，该学院因为拥有众多著名的专家而成为美国最好的学校之一。除了在大学中讲授C++之外，Shtern 还为有经验的程序人员讲授培训课程。

ISBN 7-111-10100-6
9 787111 101000

HZ BOOKS
华章图书

网上购书：www.china-pub.com
北京市西城区百万庄南街1号 100037
购书热线：(010)68995259，8006100280（北京地区）
总编信箱：chiefeditor@hzbook.com

ISBN 7-111-10100-6/TP · 2384
定价：85.00 元

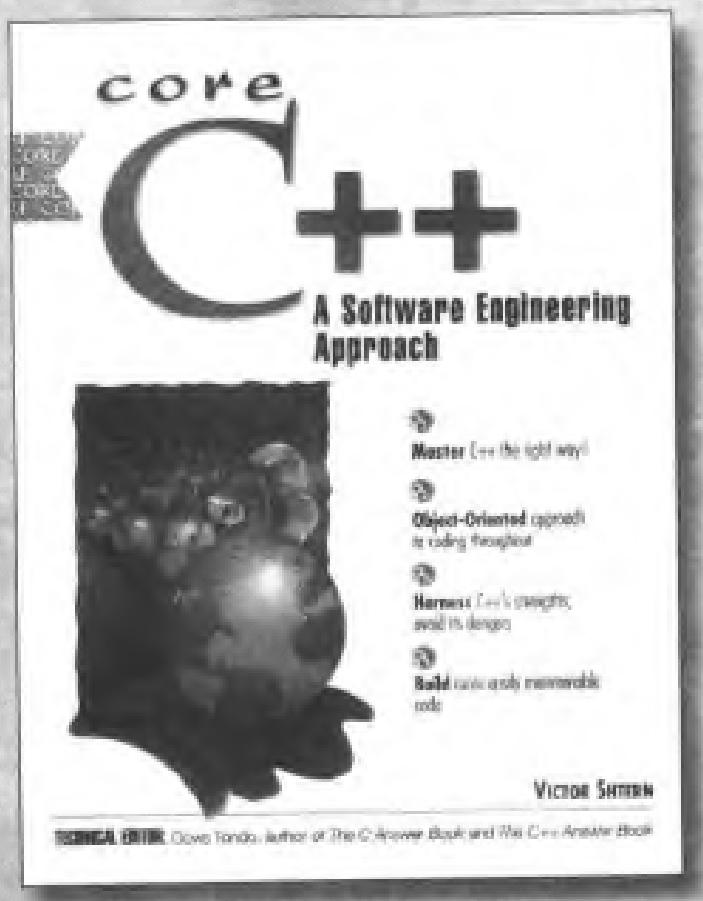
封面设计：华子

计 算 机 科 学 丛 书

C++精髓

软件工程方法

(美) Victor Shtern 波士顿大学 著 李师贤 张珞玲 刘斌 郑红 等译



Core C++

A Software Engineering Approach

机械工业出版社
China Machine Press

C++是一种大型而复杂的语言，其设计目标是作为一种通用的工程语言。

本书分4个部分共19章，不仅详细介绍了C++语言的基本语法，而且讲解了C++的高级应用（如虚函数、模板、异常等），并通过大量详尽的代码表达了有关软件工程及维护的观点。全书贯穿了面向对象程序设计思想，不断强调开发可重用的、可移植的和易维护的程序的重要性。

本书专门为希望将实际经验与C++的具体细节相结合的专业人士而编写，也是一本学习C++语言的好教材，对初学编程的读者也大有裨益。

Victor Shtern: Core C++: a software engineering approach.
Authorized translation from the English language edition published by Prentice Hall PTR.
Copyright © 2000 by Prentice Hall PTR.
All rights reserved.
Chinese simplified language edition published by China Machine Press.
Copyright © 2002 by China Machine Press.

本书中文简体字版由美国Prentice Hall PTR公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2000-3115

图书在版编目（CIP）数据

C++精髓：软件工程方法/（美）史特恩（Shtern, V.）著；李师贤等译. -北京：机械工业出版社，2002.8
（计算机科学丛书）
书名原文：Core C++: a software engineering approach
ISBN 7-111-10100-6

Ⅰ.C… Ⅱ.①史… ②李… Ⅲ.C语言-程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2002）第017182号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：温丹丹
北京市密云县印刷厂印刷·新华书店北京发行所发行
2002年8月第1版第1次印刷
787mm×1092mm 1/16·51.75印张
印数：0 001-5 000册
定价：85.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995265
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译 者 序

C++这种十分大型而且复杂的语言是作为一种通用的工程语言而创建的。今天，C++已经在商业、工程甚至实时系统中获得广泛的应用。本书通过大量详尽的代码分析，不仅详细介绍了该语言的基本语法，还讲解了C++的高级应用（如虚函数、模板、异常等）。全书贯穿了面向对象程序设计思想，不断强调开发可重用的、可移植的和易维护的程序的重要性。

市面上论述C++的书籍已经很多，但是本书的特点是通过其C++代码表达了有关软件工程及维护的观点，很少有C++方面的书能做到这一点。本书的另外一个重要的特点是其讲解的方法，它告诉读者应如何而不应如何使用C++，特别是从重用和维护的观点出发进行了介绍。本书的第三个特点是作者循序渐进地介绍各个问题，开始给出总体概述，然后再进行深入地介绍，读者不会在学习过程中因为遇到一些将依赖于后续内容的概念而无所适从。

本书专门为希望将实际经验与C++的具体细节相结合的专业人士而编写。它适用于熟悉其他编程语言并且想转用C++的人员，也可以开阔有经验的C++程序员的视野，对那些初学编程但愿意花费精力学习的读者也大有裨益。

参加本书翻译工作的有：李师贤、张珞玲、刘斌、郑红、舒忠梅、明仲、李皓、李智。唐素梅、邱璟等对本书的翻译也作出了贡献，温丹丹编辑对本书的编辑出版付出了辛勤的劳动，提出了许多建设性的意见，译者在此表示衷心的感谢。

由于译者水平所限，译文中难免有不妥之处，恳请读者不吝批评指正。

译 者

2001年12月于广州康乐园

前　言

祝贺大家找到了目前最有用的C++书籍中的一本！本书既介绍了C++的长处，也讨论了其短处。在这些方面，它比作者所看过的大量此类书籍中的任何一本都要好。

本书与其他C++书籍的区别

当然，每一位作者都会声称他所写的书是最好的书之一。本书的特点是在于表达了有关编写C++代码的软件工程及维护的观点。很少有C++方面的书能做到这一点。

为什么要强调软件工程及维护方法的重要性呢？原因在于C++语言不仅改变了人们编程的方式，而且改变了人们学习编程语言的方式。以前，我们会花一两天的时间来了解语言的基本语法，然后就可以开始编写解决一些简单问题的程序。接着，我们会继续学习更复杂的语法并解决更复杂的问题。一两个星期以后（对于一个较复杂的语言或许需要三四个星期），就可以学习完语言的全部内容并且可以成为一名“专家”了。

对于C++这种十分大型而且复杂的语言，情况却并不如此。可以把它视为C的超集，并且程序员可以很快地学会用C写出简单的程序（因此也就写出了C++程序），但是对于复杂的程序情况就不是这样。如果程序员不能很好地了解C++，就很难编写出可移植的和复杂的C++程序，编写出的代码将难以重用，因而也就难以维护。

C++是一种很庞大的语言——它是作为一种通用的工程语言而创建的，同时设计得非常成功。今天，C++已应用于商业、工程甚至实时应用中。C++语言在设计上作了很大的努力，以确保C++程序能具备以下的性能：支持动态内存管理、程序的不同部分可以相对独立。但是，即使是一个已经通过全面测试和完全没有语法错误的C++程序，在以下的三个方面还可能会存在问题：

1) 它可能会工作得慢，比同等的C程序更慢。

2) 它可能当内存使用改变时出现影响程序的内存管理错误（例如，当装入另一个程序时）；这些错误可能会使程序终止或者产生不正确的结果。

3) 它可能存在程序的不同部分之间的相互依赖，于是维护人员为了理解设计人员的意图就要花费大量的时间；一个写得不好的C++程序会比一个非面向对象的程序更难维护和重用。

上述问题有何重要性可言呢？如果只是要编写一个使用时间很短的小型程序，那么运行速度、内存管理、可维护性、可重用性等性能都不会显得很重要。程序员只需关注快速地给出问题答案的能力。如果答案不满意，可以把这一程序扔掉重编写一个新程序。如果只是这样的话，大家可以看任何一本其他的C++书籍。（当然，大家仍可购买本书并可感受C++语言及其用法的非形式的风格以及独特的洞察力）

然而，如果要参加一个开发小组，去开发一个不能轻易丢弃并且将要维护相当长时间的大型应用程序，上面三个问题都会变得十分重要。本书中提出的软件工程与维护的方法是十分有用和相当独特的。目前市面上大多数这类书籍根本没有在这方面作介绍（只要看一下索引就会知道。）。即使有，也没有明确阐明哪些是解决这类问题的技术。

本书的另外一个重要的特点是其讲解的方法。市面上许多教材都把精力集中在列举语言的特征上，而在教读者如何运用语言方面却是相当平庸的。与学习自然语言相似，当一个人读了一本法语的语法书之后，是否就会说法语了呢？作者没有学过法语，但学过英语，并且知道读语法书对流利地掌握一门语言并没有帮助。本书将告诉大家应如何而不应如何使用这一语言，特别是从重用和维护的观点出发进行介绍。

另外一个与教学相关的问题是，C++语言的许多特征是相互关联的，因此难以通过一种从简单到复杂的线性方式进行介绍。许多C++书籍的作者根本没有在这方面做过任何的尝试。他们认为这样做会“冒犯读者的智力”。实际上，他们可能在第3章提到一个概念，到第8章才作出解释，于是很容易使人感到恐惧和失望。

本书讲授C++的方法是不同一般的。它将周而复始地介绍各个主题，开始给出总体概述，然后再逐步进行深入地介绍，我们将不会学习一些将依赖于后续内容的材料。

本书作者的教学方法来源于长年在软件教学工作方面的经验。在Boston大学大都会学院，我的大部分学生在从事职业工作的同时，还坚持晚上到课堂上来进修。我也教授了许多专业课程以及场地培训课程。我十分同情学生们在理解语言的概念以及编程技术方面所做的努力。于是，我把教学经验整理为一系列周密的专题、实例、反例以及建议。我认为自己教授C++的方法是独特的，而且会使大家受益。

本书的读者对象

本书专门为希望将实际经验与C++的具体细节相结合的专业人士而编写。

如果希望通过对新技术应用的深入讨论而了解新技术的实际细节，那么本书正好能达到这一需求。

本书适用于熟悉其他编程语言并且想转用C++的人员。对于已有经验的C++程序员而言，仍会发现本书十分有用并且能够开阔视野。对于初学编程的人来说，花费一定的精力学习本书后将获得很大的收益。

本书的内容组织

在此不想像其他作者那样仔细描述书中在什么地方讨论了什么问题。现在就介绍一些陌生的术语、概念及技术不但没有什么意义，而且很可能会使人十分厌倦。这就是本书将总结放在最后一章（第19章）的理由，感兴趣的人可以先读一读，这样做可能会更有意义。

以下将会简要介绍书中可能令人感兴趣的各个部分的内容，这取决于各人的经验和背景：

- 对于一位有经验的C++程序员来说，第三、四部分中关于C++的强大功能以及编程中易犯错误的介绍将会十分有用。如果初次学习C++，初次涉足对象的概念，没有过程化程序设计、内存管理以及生成维护代码等经验，那么学习第一、二部分将是有用和有趣的。
- 对于一个想学习C++的有经验的C程序员来说，第二、三、四部分就是针对他们而写的。如果能简单地浏览一下第一部分，就会发现从软件工程以及维护的观点来讨论C是有趣的。
- 对于一位使用C、C++、Java之外的某种高级语言的有经验的程序员来说，应该从第一部分读起。

- 对于想了解编程入门的人来说，应该跳过第1章，在开始时面对这一部分将会十分抽象。首先学习第一部分的其他几章，再回到第1章，然后继续学习第二、三、四部分。

本书中使用的约定

本书中所有带编号程序中的代码均已使用多种编译程序进行了彻底的调试和测试，这些编译程序包括了Microsoft Visual C++、Borland和GNU编译程序。这些代码不加任何修改就可以运行。在这些程序之外的代码段也已通过调试和测试。它们是可运行的，但要真正运行还需遵循某种用法约定。

整本书中所列的代码或代码段都使用等宽字体。书中所给的C++术语也是如此。例如，当讨论一个C++类、其类名是“Account”时，会将其写为`Account`，与它在程序中的写法一致。

注意 用“注意”表示需引起特别注意，比如与主题有关的一个有趣的事实或者程序员在编程时需要牢记的某个事实。

警告 用“警告”表示使用时可能会引起意外的结果或者严重的错误。

提示 用“提示”给出特别有用的信息，以便节省读者的时间、强调某个有用的编程技巧或者在提高效率方面给出特别的劝告。

访问源代码的方式

在学习一门语言的时候，实践是非常重要的。学习C++而不实践，就跟学习驾驶课程而不驾驶一样。一个人可能知道许多关于驾驶的有用知识，但仍不会驾驶。在此本书强烈建议大家在学习本书的过程中使用给出的程序进行实验。程序中所给出的全部源代码可以在以下站点上获得：

```
ftp://ftp.prenhall.com/pub/ptr/c++programming.w-050/corec++
```

ftp://ftp.prenhall.com/pub/ptr/c++_programming.w-050/core_c++/

意见反馈

本书已进行了全面的审阅、仔细的编辑以及细心的校对。然而也许还会隐含一些错误。

由于我的愿望是编写一本独特的书，因而有的地方或许会是无根据的、不合理的，或许简直是错误的。或者，有些陈述是有争议并且可以讨论的。随时欢迎通过以下电子邮件地址与作者联系：shtern@bu.edu。

对于那些指出本书的排印错误，内容错误或讨论中的真正问题的人们，本书的下一个版本中将对前两位致谢。

目　录

出版者的话
专家指导委员会
译者序
前言

第一部分　C++程序设计简介

第1章　面向对象方法的优点 ……1
1.1　软件危机的起因 ……2
1.2　解决方案1：摆脱程序员 ……5
1.3　解决方案2：改进管理技术 ……7
1.3.1　瀑布模型方法 ……7
1.3.2　快速原型方法 ……8
1.4　解决方案3：设计一种复杂而完善的语言…9
1.5　面向对象方法的含义和优点 ……10
1.5.1　设计人员的工作 ……10
1.5.2　设计质量：内聚性 ……11
1.5.3　设计质量：耦合度 ……12
1.5.4　设计质量：将数据与函数绑定在一起…12
1.5.5　设计质量：信息隐藏和封装 ……14
1.5.6　设计问题：命名冲突 ……15
1.5.7　设计问题：对象初始化 ……15
1.5.8　对象的实质 ……16
1.5.9　使用对象的优点 ……17
1.6　C++程序设计语言的特征 ……18
1.6.1　C语言的目标：性能、可读性、美观和可移植性 ……18
1.6.2　C++语言的目标：与C语言向后兼容的类 ……19
1.7　小结 ……21
第2章　快速入门：C++简介 ……22
2.1　基本程序结构 ……22
2.2　预处理程序指令 ……24
2.3　注释 ……27
2.4　声明和定义 ……30
2.5　语句和表达式 ……34
2.6　函数和函数调用 ……41
2.7　类 ……47
2.8　程序开发工具的使用 ……51
2.9　小结 ……54
第3章　C++数据和表达式的使用 ……55
3.1　值及其类型 ……55
3.2　整数类型 ……57
3.2.1　整数类型修饰符 ……59
3.2.2　字符 ……62
3.2.3　布尔值 ……64
3.3　浮点类型 ……64
3.4　C++表达式的使用 ……66
3.4.1　高优先级运算符 ……66
3.4.2　算术运算符 ……67
3.4.3　移位运算符 ……69
3.4.4　按位逻辑运算符 ……70
3.4.5　关系和相等运算符 ……72
3.4.6　逻辑运算符 ……73
3.4.7　赋值运算符 ……75
3.4.8　条件运算符 ……76
3.4.9　逗号运算符 ……77
3.5　混合型表达式：隐藏的危险 ……77
3.6　小结 ……82
第4章　C++控制流 ……83
4.1　语句和表达式 ……83
4.2　条件语句 ……84
4.2.1　条件语句的标准形式 ……85
4.2.2　条件语句中的常见错误 ……88
4.2.3　嵌套条件语句及其优化 ……98
4.3　循环 ……104
4.3.1　while循环 ……105
4.3.2　do-while循环 ……111
4.3.3　for循环 ……114
4.4　C++转移语句 ……117
4.4.1　break语句 ……117
4.4.2　continue语句 ……120

4.4.3 goto语句 ……120
4.4.4 return和exit转移 ……121
4.4.5 switch语句 ……125
4.5 小结 ……127
第5章 程序员定义数据类型的聚集 ……129
5.1 同种类聚集的数组 ……129
5.1.1 作为值向量的数组 ……129
5.1.2 C++数组的定义 ……131
5.1.3 数组上的操作 ……133
5.1.4 下标正确性的检查 ……133
5.1.5 多维数组 ……136
5.1.6 字符数组的定义 ……138
5.1.7 字符数组上的操作 ……140
5.1.8 字符串函数和内存讹用 ……141
5.1.9 二维字符数组 ……144
5.1.10 插入算法中的数组溢出 ……146
5.1.11 数组类型的定义 ……150
5.2 不同种类聚集的结构 ……151
5.2.1 程序员定义类型的结构定义 ……151
5.2.2 创建和初始化结构变量 ……152
5.2.3 层次结构及其分量 ……153
5.2.4 结构变量上的操作 ……154
5.2.5 在多文件程序中定义的结构 ……155
5.3 联合、枚举和位域 ……157
5.3.1 联合 ……157
5.3.2 枚举 ……160
5.3.3 位域 ……162
5.4 小结 ……165
第6章 内存管理：栈和堆 ……166
6.1 作为合作工具的名字作用域 ……166
6.1.1 C++词法作用域 ……167
6.1.2 同一作用域中的名字冲突 ……167
6.1.3 在独立的作用域中使用相同的名字…170
6.1.4 在嵌套的作用域中使用相同的名字…171
6.1.5 循环变量的作用域 ……174
6.2 内存管理：存储类别 ……175
6.2.1 自动变量 ……176
6.2.2 外部变量 ……178
6.2.3 静态变量 ……182
6.3 内存管理：堆的使用 ……186
6.3.1 作为类型变量的C++指针 ……187
6.3.2 堆的内存分配 ……191
6.3.3 数组和指针 ……194
6.3.4 动态数组 ……197
6.3.5 动态结构 ……207
6.4 磁盘文件的输入和输出 ……215
6.4.1 输出到文件 ……215
6.4.2 从文件输入 ……217
6.4.3 输入/输出文件对象 ……221
6.5 小结 ……223

第二部分 用C++进行面向对象的程序设计

第7章 使用C++函数编程 ……225
7.1 作为模块化工具的C++函数 ……226
7.1.1 函数声明 ……227
7.1.2 函数定义 ……228
7.1.3 函数调用 ……228
7.2 参数的提升和类型转换 ……230
7.3 C++中函数的参数传递 ……231
7.3.1 按值调用 ……231
7.3.2 按指针调用 ……232
7.3.3 C++中的参数传递：按引用调用 …238
7.3.4 结构 ……241
7.3.5 数组 ……246
7.3.6 类型转换的进一步讨论 ……249
7.3.7 从函数返回值 ……251
7.4 内联函数 ……255
7.5 有缺省值的参数 ……257
7.6 函数名重载 ……261
7.7 小结 ……267
第8章 使用函数的面向对象程序设计 ……268
8.1 内聚性 ……270
8.2 耦合度 ……270
8.2.1 隐式耦合度 ……271
8.2.2 显式耦合度 ……274
8.2.3 如何降低耦合度 ……277
8.3 数据封装 ……281
8.4 信息隐藏 ……286
8.5 一个有关封装的大型例子 ……291

8.6 用函数实现封装的不足 ……299
8.7 小结 ……301
第9章 作为模块单元的C++类 ……303
9.1 基本的类语法 ……304
9.1.1 绑定数据与操作 ……304
9.1.2 消除名字冲突 ……307
9.1.3 在类之外实现成员函数 ……310
9.1.4 不同存储方式的类对象的定义 ……313
9.2 对类成员的控制访问 ……314
9.3 对象实例的初始化 ……319
9.3.1 作为成员函数的构造函数 ……320
9.3.2 缺省构造函数 ……321
9.3.3 拷贝构造函数 ……323
9.3.4 转换构造函数 ……325
9.3.5 析构函数 ……326
9.3.6 构造函数和析构函数的调用时间 …330
9.3.7 类作用域和嵌套作用域中的名字覆盖 ……331
9.3.8 用运算符和函数调用的内存管理 …333
9.4 在客户代码中使用返回的对象 ……336
9.4.1 返回指针和引用 ……336
9.4.2 返回对象 ……338
9.5 关于const关键字的讨论 ……340
9.6 静态类成员 ……345
9.6.1 用全局变量作为类特性 ……345
9.6.2 关键字static的第四种含义 ……347
9.6.3 静态数据成员的初始化 ……348
9.6.4 静态成员函数 ……348
9.7 小结 ……351
第10章 运算符函数：另一种好设计思想…353
10.1 运算符重载 ……353
10.2 运算符重载的限制 ……360
10.2.1 不可重载的运算符 ……360
10.2.2 返回类型的限制 ……361
10.2.3 参数个数的限制 ……363
10.2.4 运算符优先级的限制 ……364
10.3 把重载运算符作为类成员 ……364
10.3.1 用类成员取代全局函数 ……364
10.3.2 在链式操作中使用类成员 ……366
10.3.3 使用const关键字 ……368
10.4 案例分析：有理数 ……370
10.5 混合参数类型 ……377
10.6 友元函数 ……383
10.7 小结 ……394
第11章 构造函数与析构函数：潜在的问题 ……395
11.1 对按值传递对象的深入讨论 ……396
11.2 非数值类的运算符重载 ……400
11.2.1 String类 ……401
11.2.2 堆内存的动态管理 ……402
11.2.3 保护客户代码中的对象堆数据 ……406
11.2.4 重载的串接运算符 ……406
11.2.5 防止内存泄漏 ……408
11.2.6 保护程序的完整性 ……409
11.2.7 如何由此及彼 ……413
11.3 对拷贝构造函数的深入讨论 ……414
11.3.1 完整性问题的补救措施 ……415
11.3.2 拷贝语义和值语义 ……419
11.3.3 程序员定义的拷贝构造函数 ……420
11.3.4 按值返回 ……423
11.3.5 拷贝构造函数的有效局限性 ……427
11.4 赋值运算符的重载 ……427
11.4.1 系统提供的赋值运算符的问题 ……427
11.4.2 重载的赋值：内存泄漏 ……428
11.4.3 重载的赋值：自我赋值 ……429
11.4.4 重载的赋值：链表达式 ……430
11.4.5 程序性能的考虑 ……433
11.4.6 第一种补救措施：更多的重载 ……433
11.4.7 第二种补救措施：按引用返回 ……434
11.5 实用性的考虑：实现什么函数 ……435
11.6 小结 ……438

第三部分 使用聚集和继承的面向对象程序设计

第12章 复合类的优缺点 ……439
12.1 用类对象作为数据成员 ……440
12.1.1 C++类复合的语法 ……442
12.1.2 访问类数据成员的数据成员 ……443

12.1.3 访问方法参数的数据成员 ……445
12.2 复合对象的初始化 ……446
12.2.1 使用组件的缺省构造函数 ……448
12.2.2 使用成员的初始化列表 ……453
12.3 具有特殊属性的数据成员 ……457
12.3.1 常量数据成员 ……458
12.3.2 引用数据成员 ……459
12.3.3 用对象作为其类自身的数据成员 …461
12.3.4 用静态数据成员作为其类自身的数据成员 ……463
12.4 容器类 ……465
12.4.1 嵌套类 ……478
12.4.2 友元类 ……480
12.5 小结 ……482
第13章 如何处理相似类 ……484
13.1 相似类的处理 ……485
13.1.1 把子类的特征合并到一个类中 ……486
13.1.2 把保持程序完整性的任务推向服务器 ……488
13.1.3 为每种服务器对象建立单独的类 …492
13.1.4 使用C++的继承去链接相关类 ……495
13.2 C++继承的语法 ……497
13.2.1 基类的不同派生模式 ……498
13.2.2 定义和使用基类对象和派生类对象 ……501
13.3 对基类和派生类服务的访问 ……503
13.4 对派生类对象的基类成员的访问 ……506
13.4.1 公共继承 ……507
13.4.2 受保护继承 ……511
13.4.3 私有继承 ……515
13.4.4 调整对派生类中基类成员的访问 …517
13.4.5 缺省继承模式 ……518
13.5 在继承下的作用域规则和名字解析 …520
13.5.1 名字重载与名字隐藏 ……522
13.5.2 派生类所隐藏的基类方法的调用 …525
13.5.3 使用继承改进程序 ……529
13.6 派生类的构造函数和析构函数 ……532
13.6.1 在派生类构造函数中的初始化列表 ……535
13.6.2 继承中的析构函数 ……537
13.7 小结 ……539
第14章 在继承和复合之间进行选择 ……541
14.1 选择代码重用的方法 ……542
14.1.1 类之间的客户–服务器关系的例子…542
14.1.2 运用智力的重用：重做 ……545
14.1.3 借助服务重用 ……547
14.1.4 通过继承的代码重用 ……550
14.1.5 以重新定义函数的方式继承 ……554
14.1.6 继承和复合的优缺点 ……556
14.2 统一建模语言 ……557
14.2.1 使用UML的目的 ……557
14.2.2 UML基础：类的表示 ……560
14.2.3 UML基础：关系的表示 ……561
14.2.4 UML基础：聚集和泛化的表示 ……562
14.2.5 UML基础：多重性的表示 ……563
14.3 案例分析：一个租赁商店 ……565
14.4 类的可见性和任务划分 ……580
14.4.1 类的可见性及类之间的关系 ……580
14.4.2 将任务推向服务器类 ……582
14.4.3 使用继承 ……584
14.5 小结 ……585

第四部分 C++的高级应用

第15章 虚函数和继承的其他高级应用 …587
15.1 非相关类之间的转换 ……589
15.1.1 强类型与弱类型 ……591
15.1.2 转换构造函数 ……592
15.1.3 指针或引用之间的转换 ……593
15.1.4 转换运算符 ……594
15.2 通过继承相关的类之间的转换 ……594
15.2.1 安全转换与不安全转换 ……595
15.2.2 对象的指针与引用的转换 ……599
15.2.3 指针与引用参数的转换 ……607
15.3 虚函数 ……612
15.3.1 动态绑定：传统方法 ……615
15.3.2 动态绑定：面向对象的方法 ……622

15.3.3 动态绑定：使用虚函数 ……629
15.3.4 动态绑定与静态绑定 ……633
15.3.5 纯虚函数 ……636
15.3.6 虚函数：析构函数 ……639
15.4 多继承：多个基类 ……640
15.4.1 多继承：访问规则 ……642
15.4.2 类之间的转换 ……643
15.4.3 多继承：构造函数和析构函数 ……644
15.4.4 多继承：二义性 ……645
15.4.5 多继承：有向图 ……647
15.4.6 多继承：有用吗 ……648
15.5 小结 ……649
第16章 运算符重载的高级应用 ……651
16.1 运算符重载简介 ……651
16.2 一元运算符 ……659
16.2.1 ++和--运算符 ……659
16.2.2 后缀重载运算符 ……666
16.2.3 转换运算符 ……668
16.3 下标和函数调用运算符 ……674
16.3.1 下标运算符 ……675
16.3.2 函数调用运算符 ……683
16.4 输入/输出运算符 ……687
16.4.1 重载运算符>> ……688
16.4.2 重载运算符<< ……691
16.5 小结 ……693
第17章 模板：另一个设计工具 ……694
17.1 类设计重用的一个简单例子 ……694
17.2 模板类定义的语法 ……701
17.2.1 模板类说明 ……702
17.2.2 模板实例化 ……703
17.2.3 模板函数的实现 ……704
17.2.4 嵌套模板 ……710
17.3 多参数的模板类 ……711
17.3.1 多类型参数 ……711
17.3.2 带有常量表达式参数的模板 ……713
17.4 模板类实例之间的关系 ……716
17.4.1 作为友元的模板类 ……717
17.4.2 嵌套模板类 ……719
17.4.3 带静态成员的模板 ……722
17.5 模板的规则说明 ……724
17.6 模板函数 ……728
17.7 小结 ……729
第18章 带异常处理的程序设计 ……730
18.1 异常处理的一个简单例子 ……730
18.2 C++异常的语法 ……736
18.2.1 抛出异常 ……737
18.2.2 捕获异常 ……738
18.2.3 声明异常 ……743
18.2.4 重新抛出异常 ……745
18.3 类对象的异常 ……748
18.3.1 抛出、声明与捕获对象的语法 ……749
18.3.2 使用带异常处理的继承 ……752
18.3.3 标准异常库 ……756
18.4 类型转换运算符 ……757
18.4.1 static_cast运算符 ……757
18.4.2 reinterpret_cast运算符 ……761
18.4.3 const_cast运算符 ……761
18.4.4 dynamic_cast运算符 ……764
18.4.5 typeid运算符 ……766
18.5 小结 ……767
第19章 总结 ……769
19.1 作为传统程序设计语言的C++ ……769
19.1.1 C++内部数据类型 ……769
19.1.2 C++表达式 ……771
19.1.3 C++控制流 ……772
19.2 作为模块化语言的C++ ……772
19.2.1 C++聚集类型之一：数组 ……773
19.2.2 C++聚集类型之二：结构、联合和枚举类型 ……774
19.2.3 作为模块化工具的C++函数 ……774
19.2.4 C++函数的参数传递 ……776
19.2.5 C++中的作用域与存储类别 ……776
19.3 作为面向对象语言的C++ ……778
19.3.1 C++类 ……778
19.3.2 构造函数、析构函数和重载运算符 ……778

19.3.3 类复合与继承 ……779
19.3.4 虚函数与抽象类 ……781
19.3.5 模板 ……782
19.3.6 异常 ……782
19.4 C++及其竞争对手 ……783
19.4.1 C++与传统的语言 ……783
19.4.2 C++与Visual Basic ……783
19.4.3 C++与C ……784
19.4.4 C++与Java ……785
19.5 小结 ……786
索引 ……787

第一部分 C++程序设计简介

本书的第一部分是关于C++编程的基础知识。每个人都知道，C++是一种面向对象的程序设计语言，但这意味着什么呢？为什么使用面向对象的程序设计语言，要优于传统的非面向对象的程序设计语言呢？在编程的时候，用户应该注意什么问题，才可以得到面向对象的好处呢？通常，人们不假思索地接受了面向对象的方法，但不一定能很有效地运用此方法。

第1章回答了以上的问题，并围绕着如何将程序分割成若干部分而展开讨论。一个大型程序是由一组相对独立而又相互通信和协作的组件构成的。但是，如果把应该放在一起的部分分开了，就会引起程序各部分之间过多的通信和依赖，于是代码会变得难以重用和维护。反之，如果将应该分离的组件放在一起，那么可以想像，其结果是使代码复杂化和模糊不清，同样也会难以维护和重用。

使用对象并没有什么新奇，也并没有直接的好处。但是使用对象的程序会避免以下两种危险的出现：将应该放在一起的部分分开和将应该分开的部分放在一起。第1章讨论了这些问题，它说明了可以用面向对象的方法解决什么问题，以及如何用面向对象的方法解决这些问题。

第2章，简单地介绍了包括对象等概念在内的C++语言。这是概括性的介绍（我们必须学习书中其他章节以了解其具体的细节）。然而，本章介绍的内容已足以使我们写出简单的C++程序，并且为将来仔细地研究C++的优缺点做好了准备。

第一部分的其他几章讨论了该语言基本的非面向对象特征。正如我在第1章所作的承诺，这里特别注意编写一些可重用和可维护的代码。对于每一种C++结构，我都要说明应该如何使用以及不要如何使用。尽管在这一部分还没有讨论对象，但是内容还是相当复杂的，尤其是在第6章。毕竟，C++是一种复杂的语言。当读者对某些问题的讨论感到困惑时，可以跳过那些部分，等将来再回头阅读，如果时间充裕还可以用更多时间关注编码的细节。

第1章 面向对象方法的优点

面向对象方法覆盖了所有软件开发领域。它开创了一个软件开发的新局面，并带来了许多新的好处。许多开发人员采用面向对象的方法是以为这些好处会一直存在，并且是非常重要的。然而，它们是什么？这些好处能因为用户不用函数而改用对象就自动出现吗？

本章，将首先说明使用面向对象方法的原因。如果读者是有经验的软件专业人员，可以跳过这些内容，直接阅读有关面向对象方法给软件开发带来好处的原因。

如果读者是新手，那么就应该阅读有关软件危机以及其解决方法的讨论，以便能了解本书倡导的编程技术的背景。我们将会更好地了解C++编程哪些方面有利于改善程序质量，而哪些方面却不能，以及其原因。

注意到在业界中使用的许多低质量的C++代码是很重要的。许多程序员以为只要使用C++和类就自然会有很多好处，不必理会这些代码和类是什么。事实上，这是不正确的。不幸的是，许多C++的书籍都只是集中精力去说明C++的语法而忽略了对C++代码质量的讨论。当开发人员不清楚使用C++代码的目的时，他们就会将面向对象的方法与传统的方法相混淆，于是开发出来的程序并不比传统的使用C、PL/I（或任何一种读者所喜欢的）语言的程序更优，也同样难以维护。

1.1　软件危机的起因

面向对象方法是解决业界中所谓的软件危机的一种手段。这些危机问题包括：经常性的费用超支、项目的延期或撤消、系统功能的不完善以及软件错误。软件错误带来的后果包括：给用户操作带来的不方便，以至由于错误地记录业务数据而造成的不可估量的经济损失。最终，软件错误还构成对人类生存的威胁和任务的失败。改正软件错误是十分昂贵的，通常会造成巨额的软件花费。

大多数专家认为造成软件危机的原因是缺乏一种标准的开发技术，因为软件产业仍十分年轻。其他的工程技术行业相对要成熟得多，它们已建立了自己的技术、方法论和标准。

例如在建筑业，广泛地使用了多种标准和建筑法规。在设计和建造过程的每一阶段都有详细的规程。项目的每一位参与者都能明确项目的预期目标、以及如何证明是否达到了给定的质量标准。标准的各种质量保证都是可以验证和强制执行的。消费者保护法保护了消费者的利益不会受到奸诈的或愚蠢的经营者的侵犯。

对于新兴的产业，例如汽车工业和电气工程行业，也是同样的情况。在所有这些人们所从事的行业中，普遍存在着标准、公认的开发和制造方法、制造商的质量保证以及消费者保护法。这些已确立的产业的另一个重要的特征是，它们的产品都是通过现成的部件组装起来的。这些部件是标准化的、经过严格的检验并且是大规模生产的。

对软件产业的上述方面进行比较可知道，软件产业几乎没有什么可值得一提的标准。当然，该行业中的专门组织正试图提供从编写软件测试到人机接口的规范等方面的标准。但这些标准只解决了表面的一些问题，目前还没有被普遍接受、执行及遵守的软件开发的过程与方法。市场上所谓的软件质量保证只是一个玩笑而已。如果软件提供商能够负责其产品的出错费用的话，获得该产品的消费者就十分幸运了。市场上不存在返还规则：一旦用户启封某一软件产品，就丧失了退款的权利。

产品是通过手工制造的。没有现成的商品化的组件。没有一种普遍认可的组件以及软件产品应该是怎么样的协定。在对微软公司的诉讼案中，美国政府提出的论据就在操作系统及其组件的定义上：浏览器是操作系统的一个部分还是另外的一个应用软件，例如像字处理、电子表格、日程安排等软件一样的应用软件。操作系统对计算机的重要性就像汽车的点火系统一样（甚至比这更重要）。但读者可以想像关于一个点火系统组成的法律争论吗？我们都知道如果有技术需求，就有必要使化油器成为点火系统的一个部分。但当技术改进了之后，就会取消化油器，这并不会引起公众的争论。

年轻的软件产业确实正在为社会贡献着自己的力量。希望造成这些阴暗面的因素将在不远的将来消失。然而，软件产业的年轻并没有阻止它成为一个在经济领域有着几十亿美元收入的重要角色。因特网改变了我们从事商业和搜索信息的途径。同样，它也改变了股票市场

的运作方式。

预言家预示“2000年”问题将会成为经济方面的主要威胁。讨论这种担心是否合理本身并不重要。重要的是从其能力的角度来看，软件产业已经变得相当成熟了。当一个软件问题可能会造成对整个西方社会的破坏时，就意味着这一产业有着十分重要的社会地位。然而，该产业的技术仍落后于其他产业，主要的原因在于软件开发过程的性质。

很少有如此简单的软件系统，可以由一个人单独地对它作出规格说明：根据规格说明建立系统，把它用于既定的目的，并在需求改变或者发现错误时维护系统等。如果有这样的系统，那么一定只是一个用途有限、生命周期很短的简单系统。这样一种系统在必要时很容易推倒重来，所耗费的时间和金钱都不值得一提。

大多数软件系统所具有的特征是完全不同的。它们都很复杂，难以由一个人单独完成。必须有几个人（通常是许多人）共同参与到系统的开发过程中，相互协作地开展工作。当工作分配给若干个人共同完成时，我们希望能将系统分成若干个相互独立的部分，以便每一个开发人员能够独立地完成分配给他的那部分。

例如，我们可以将软件系统的功能分成单独的一些操作（例如发一份订单、增加一个顾客、删除一个顾客等）。如果这些操作同样过于复杂，那么让一个程序员来完成将会花费相当长的时间。于是，我们将每个操作分成若干个步骤和子步骤（例如认证顾客、输入订单数据、验证顾客的信用度，等等），并把每一个步骤交给一个程序员来完成（如图1-1所示）。

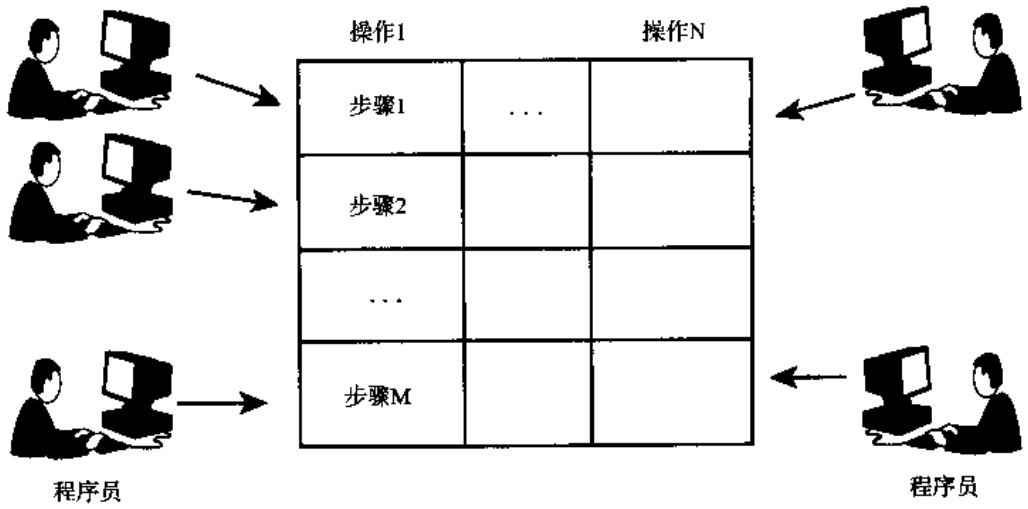


图1-1 将系统分成若干组件

我们的目的就是要使系统的组件之间相互独立，以便能由不同的人独立地完成它们。实际上，这些分开的组件并不相互独立。毕竟，它们是同一个系统的各个部分；因此，它们需要相互调用，或者要共享同样的数据结构，或者要实现同一个算法的不同步骤。由于不同的开发人员所实现的部分并不是相互独立的，因此每个开发人员必须相互合作：他们需要书写备忘录、生成设计文档、向其他人发电子邮件以及参加相关的会议、审议设计方案或检查代码。这些就是可能隐藏错误的地方——某些问题会被误解，某些事情可能会被遗忘，某些问题可能在相关决策已经改变后仍未及时修改。

复杂系统的设计、开发、测试要经过一个很长的时间才能完成，这样的系统是昂贵的，有些则是非常昂贵。许多用户依赖于它们进行工作。当需求改变、发现错误、或者不能满足

需求时，这种系统不能随便更换，因为已花费了大量的投资而不能丢弃。

这些系统必须进行维护，于是代码一定要修改。代码中某一个部分的修改总会影响到代码的其他地方，这就需要多次的修改。如果没有注意到这种依赖性（有时就会忽略），系统就会工作异常，直到这些代码（以及这些代码在其他部分的进一步影响）修改好为止。尽管维护这些复杂系统同样十分昂贵，以及同样不可避免地存在错误，但是由于开发这些系统已花费了巨额的代价，因此对它们要进行长期维护。

这里再次提到了“2000年”问题。许多人很奇怪为什么这些程序员只使用最后两位数来表示年份。大家会问，“这些程序员到底生活在什么世界？”“难道他们不明白从1999年转换到2000年意味着什么吗？”是的，这确实奇怪。但不是对程序员目光短浅的惊讶，而是奇怪这些程序员在20世纪70、80年代设计的系统会如此经久耐用。这些程序员与任何一位Y2K专家一样，很清楚从1999年转换到2000年意味着什么。但他们想不到的是20世纪70、80年代的程序直到2000年还会有人在用。

是的，今天许多组织仍把大量的金钱投入到旧软件的维护方面，原因就是这些系统太复杂了，以致重建这些系统可能花费更大，还不如选择继续维护他们。

大多数软件系统的最本质的特征就是复杂性。问题域本身是复杂的，管理软件的开发过程也是复杂的，而由单独部件手工地构造软件的技术已不足以解决这一复杂问题。

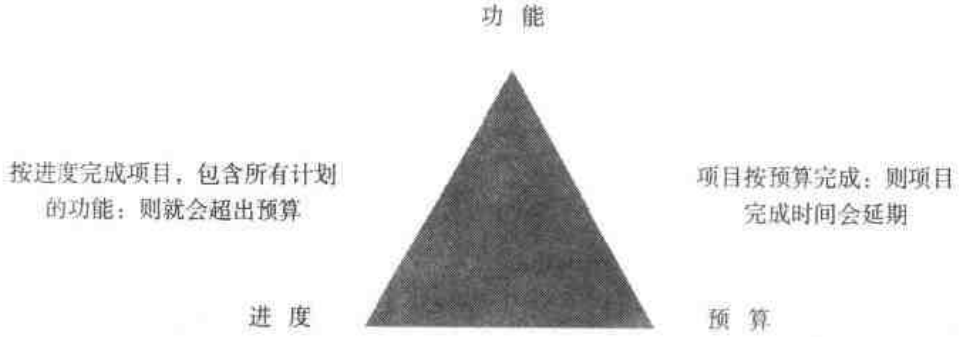
系统任务（在此我们将其称为“问题域”）的复杂性在于很难简单明了地向用户描述系统的功能，不管这是一个工程问题，或者是一个商业操作，或者是一个大型的商品化包装软件，或者是一个因特网应用软件。潜在的系统用户（或者市场上的专业人士）都觉得难以用一种容易使软件开发人员理解的形式表达他们的需求。来自不同部门用户的需求往往会相互矛盾，发现并调和这些矛盾是一项困难的工作。另外，用户和市场的需求是随着时间不断变化的，甚至就在形成需求的过程中，由于对系统实现细节的讨论会引出新的思想，于是需求就会随之发生变化。因此程序员通常抱怨的就是用户（以及市场上的专业人士）不知道他们的真正需求是什么。目前还很少有获取系统需求的工具。这就是通常把需求通过一大堆带有图表的文档来表达的原因。通常这些文档的结构相当混乱，因而难以理解；需求中许多描述是含糊的、不完整的、自相矛盾的或者还需要作进一步的解释。

管理开发过程的复杂性源自需要在许多专业开发人员之间进行协调，特别是当这组完成系统的不同部分的开发人员不在同一个地点，却需要相互交流信息以及对相同的数据进行处理时。例如，如果系统的一个部分产生的数据是用码作为长度单位来表示的话，那么使用这些数据的系统的另一部分就不能认为数据是用米作为长度单位的。这种一致性的规定是简单的，但却因为有许多这样的规定，以至要时刻记住它们是十分困难的。这就是为何让更多的人加入某个项目却未必对项目有帮助的原因。新加入的人员必须接手做目前已有人正在做的某些任务。通常的情况是，要么让新加入的人员接手去做一些计划要完成的某部分项目，要么将该部分项目再细分为更小的部分并交给新加入的人员去完成。

新加入的人员不能马上做出成绩。因为他们必须先了解出现在的开发小组已做出的项目决策。而现在的开发小组的进度也会减慢，因为他们必须牺牲做出成果的时间以便与新加入的人员充分沟通，使他们了解项目的具体情况。

将手工编制的每个模块组装起来也会带来以下的问题：这很费时间并且容易出错。测试是艰难的、手工进行的，也是不可靠的。

在我到美国时，我的老板John Convey用以下的方法向我描述了目前软件开发的状况：他画了一个三角形，其顶点分别代表进度、预算以及系统功能等项目特性（如图1-2所示）。他说，“我们不能同时获得这三样特性，实际上只能放弃某一方面的特性。如果要按预算完成项目的全部功能，那么就不可能准时完成项目，就会申请延期。如果要按进度完成项目的全部功能，那么很可能超过了原定的预算，于是要申请更多的资源。如果要按进度和预算来实现系统（尽管不经常会，但也有这种可能），那么就要去掉一些不太重要的功能的实现，只完成认为是最重要的那些部分。”



项目按预算和进度完成，则它不能完成在
项目开始时许诺的所有功能

图1-2 关于软件项目的难解三角形

用三角形描述的问题已困扰了软件产业多年。最早是在1968年提出了软件危机一词。目前业界研究了一些解决该问题的方法。以下让我们简单地了解一下这些已有的解决方案。

1.2 解决方案1：摆脱程序员

过去，在计算机系统的成本中，硬件方面的成本占了相当大的一部分，而软件方面的成本只占其中相当小的部分。系统开发的瓶颈主要在程序员与软件用户之间的交流上，这时，用户总是要向程序员解释他们以及将来的系统要完成什么样的工作。

程序员总是不能很好地正确理解用户的意图，因为他们所接受的只是数学等方面的专业训练，却没有接受过商业、工程等方面的专业训练。他们不了解商业和工程方面的术语。在另一方面，商业和工程的管理者也不熟悉软件设计和编程方面的术语。因此，当程序员试图表达他们所了解的需求时，经常会出现交流上的障碍。

类似地，程序员经常会误解用户的目标、前提以及限制条件。于是，用户就无法得到他们所希望得到的东西。

这时，解决软件危机的最好方法似乎是摆脱程序员，让商业和工程管理者直接编写应用软件，不必通过程序员去实现。然而，那时的程序员仅仅可以使用机器语言和汇编语言进行编程。这些语言要求使用者十分熟悉计算机的体系结构以及整个机器指令集。这对于本身并没有经过计算机专业训练的商业和工程管理者来说是太困难了。

为了解决这一问题，就需要一个能够方便、快速地编写软件的程序设计语言。这些语言应该使用简单，以便让工程师、科学家以及商业管理者都能够编写自己所需的软件而不必再向程序员解释他们的需求。

FORTRAN和COBOL就是最早出现的这类专为科学家、工程师和商业管理者而设计的语言。他们只要使用这些语言编程就不需要再与程序员进行交流了。

这种解决方法是有效的。许多科学家、工程师以及商业管理者成功地学会了编写他们所需的程序。一些专家预言，编程这一职业不久将会消失。但是，这种解决方法只是在编写小型程序时才有用，这些小型程序可以由单个人完成需求分析、设计、实现、文档化、使用以及维护等工作。此方法只对不需要很多开发人员协作开发的软件并且不需要长年维护的系统有效。这种程序的开发人员在项目的开发过程中不需要与他人进行合作。

实际上，图1-3只适用于小型程序。对于更大的程序的情况用图1-4描述更合适。现实的情况确实是这样的：程序开发人员之间的交流比程序开发人员与用户之间的交流显得更为重要。不论这些程序开发人员是职业程序员、职业工程师、科学家还是管理人员，正是他们之间的交流问题导致了误解、不一致性甚至是错误。

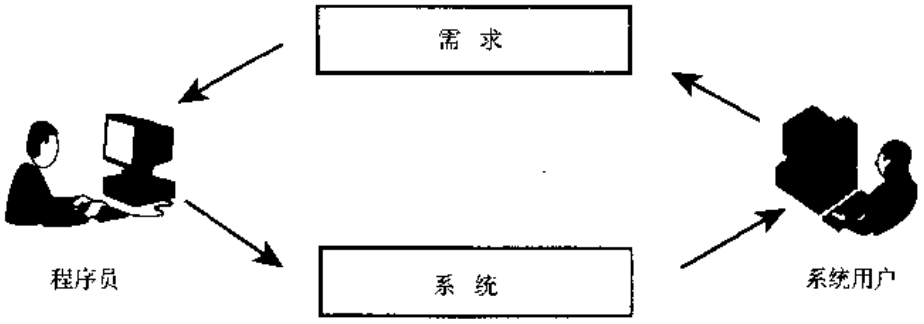


图1-3　用户与程序员之间的交流障碍

图1-4是一个过于简化的描述，其中只有若干个用户确定系统的需求和评价系统的有效性。对于大多数软件项目来说，会有许多用户（包括市场营销代表、销售人员）确定系统的需求，会有若干人进行项目评价，一般来说他们不是同一批人。于是在确定系统的需求与评价系统的质量之间又会出现不一致性和误解，这进一步加深了程序员交流问题的严重性。这种情况尤其容易出现在开发一个功能与某个现有系统功能相似的新系统的时候。此时，不同的开发人员会对同一个问题有着不同的理解。

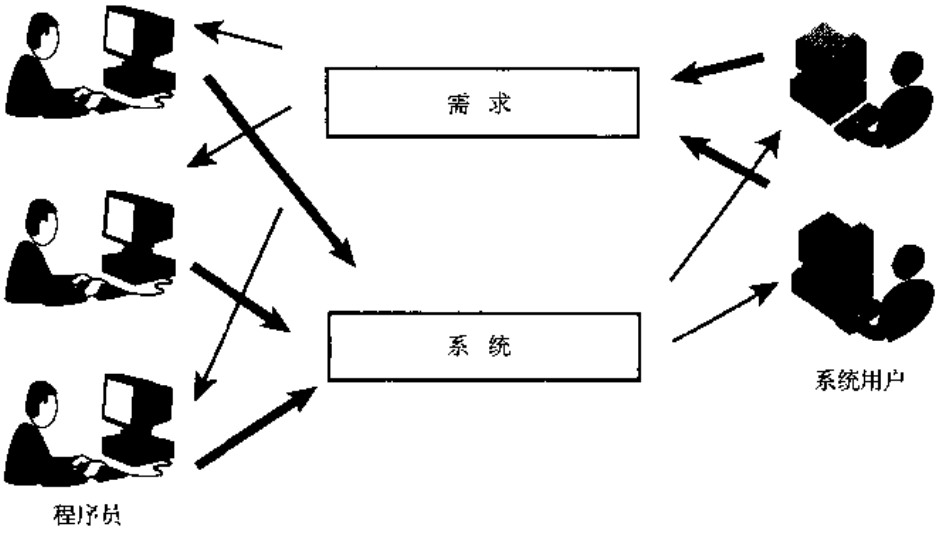


图1-4　程序开发人员之间的交流障碍

另一个企图摆脱程序员的途径是使用超级程序员。其思想很简单。如果一般的程序员不能开发出连接起来没有错误的程序组件，那么就去找一个有足够能力可以单独开发整个程序的人。尽管超级程序员的工资会比一般的程序员要高，但这是值得的。如果由同一个人来开发同一个系统的不同部分，那么就会减少不一致性问题的出现，就会较少出错，即使有错也较易改正。

实际上，超级程序员是不可能独自工作的，许多简单的工作只需由薪水很低的普通人员就可以完成。因此，超级程序员的工作必须得到技术员、资料员、测试员、文书等人的支持。

这种方法所取得的成功是有限的。实际上，每一个开发项目都无法确保成功，即解决图1-2的限制，使开发能够按进度和预算完成所需的功能。实际的情况是，超级程序员与其支持人员之间的交流又受到了支持人员能力的限制。

另外，超级程序员也不适于做长期的维护工作，他们可能会转去负责其他项目，可能会被提拔到更高一级的职位而不再编程，还可能为了接受新的挑战而转换到其他组织工作。当普通的程序员接手去维护超级程序员所编写的程序时，就会遇到与维护一般程序员编写的程序同样多甚至是更多的麻烦，因为超级程序员通常会编写一些相当简洁的文档：对于一个超级程序员来说，即使是复杂的系统也是相当简单的，于是不需要编写详细的文档。

目前，没有人能够承诺我们不需要程序员就可以开发软件系统。软件产业正在研究只须由能力一般的人员就可以开发出高质量软件系统的技术。解决的方法是使用管理技术。

1.3 解决方案2：改进管理技术

随着硬件费用的不断下降，软件开发和维护的费用占了整个计算机系统费用的绝大部分。开发一个昂贵的软件系统就意味着要投入一笔不可以轻易放弃的巨资。因此尽管维护费用相当昂贵，还是要对昂贵的系统进行长期的维护。

硬件能力的不断增强开阔了新的应用范围，这进一步增大了编码的复杂性和软件的费用（开发和维护两者的费用也随之增大）。

这就改变了我们在软件开发过程中优先考虑的事情。原来寄希望于由非常优秀的少数人来解决问题的方法已变得不可能，于是，业界转向研究在一般的用户和开发人员之间，特别是在负责项目不同部分的开发人员之间进行沟通的管理方法。

为了使用户和开发人员之间便于沟通，业界使用了以下两种管理技术：

- 将开发过程划分为各个独立阶段的瀑布模型方法。
- 部分实现用户需求并及早反馈信息的快速原型方法。

1.3.1 瀑布模型方法

在管理编程开发的过程中，有许多瀑布模型方法的变种。所有的方法都将整个开发过程划分为顺序进行的若干个阶段。一种典型的阶段划分可能包括：需求定义、系统分析、体系结构设计、详细设计、实现和单元测试、集成测试、验收测试以及维护。通常由一个单独的专门开发小组负责其中一个开发阶段。经过一段时间对系统功能的试运行以后，就会提出对系统的新的或修改的需求，于是，上述各个阶段又会不断重复进行。

阶段之间的转换标志是每个阶段所生成的详细文档。理想情况下，每个阶段所生成的文档有两种用途：对前一个阶段做出反馈，评价其正确性以及作为下一个开发阶段的输入文档。

这一工作可以非正式地进行：将文档分发给相关的人员；也可以正式地进行：召开每个开发小组的代表以及用户代表的会议，以进行复审。

例如，在需求定义结束时，需求文档就可以交给项目组织者和用户代表以征求意见，还可以作为系统分析阶段的输入文档。同样，系统分析阶段所生成的详细系统规格说明既可以反馈给用户，也可以作为设计阶段的输入文档。这是一种理想的情况。实际上，提出反馈信息的人员通常由于其他工作的限制，而只是拿出很少的时间来提出反馈意见。这就难以达到开发过程中的质量控制要求。

另外，随着项目的进一步开展，就更难从用户那里得到有用的反馈信息，因为所使用的表达方式越来越面向计算机，用户越来越不熟悉所使用的图表等记号。于是每个阶段的复审工作就退化成一个盖章的动作而已。

这种方法的优点是所使用的良好的结构定义，它清楚地规定了每个阶段的每一个开发人员的角色以及所需的文档。已经出现了许多针对不同的阶段进行项目设计以及阶段工作和费用评价的具体方法和工具。这一点对于大型项目来说是十分重要的，特别是在我们希望保证项目朝着所希望的方向发展的时候。在一个项目中所积累的经验对以后相似项目的开发会十分有帮助。

这种方法的缺点是：过度的形式化、隐藏在群体背后的个人责任问题、效率不高以及反馈机制的延时。

1.3.2　快速原型方法

快速原型方法是一种相反的方法。为了更好地得到用户的反馈信息，它消除了各个严格的阶段规定。代替原来规格说明书的是可以展示给用户的系统原型。用户通过尝试使用原型就可以更早地提出比瀑布模型更多的反馈意见。这似乎是非常好的一种方法，但对于开发一个大型的系统来说却未必有效，因为根本就不能很快建立一个大型系统的原型，而且会使建立整个系统的复杂性和费用都增大。试用原型的用户为此增加了更直接的责任负担，他们可能会缺乏操作该系统、测试系统以及给开发人员提供系统反馈意见等方面的技巧（或时间）。

这种方法在定义系统的用户界面，包括菜单、对话框、文本域、控制按钮以及其他的人机交互的组件时最有效。许多开发组织把两种开发方法结合在一起使用并取得了成效。但是同样还是无法消除系统不同部分的开发人员之间在交流沟通上的障碍。

为了解决开发人员之间的沟通问题，已开发了一些形式化的“结构化”技术，并取得了一定的成功。为了编写系统需求和规格说明，使用了结构化的英语（或者开发人员所熟悉的语言）来方便理解问题的描述以及确切说明问题的各个部分。为了定义系统的总体结构及组成部分，普遍使用数据流图以及状态转换图等技术，使结构设计的结果更通用。在详细设计阶段，出现了各种形式的流程图和结构化伪代码，从而方便了对算法和代码各部分之间相互联系的理解。在实现阶段，使用了结构化程序设计的方法。结构化程序设计在代码中限制使用跳转，因此使代码更易于让人理解（至少它能大大降低理解代码的复杂性）。

没有必要在此详细地介绍这些技术。这种形式化的管理和文档化技术是十分有用的。如果没有它们，情况会更糟。但是它们不能从根本上消除软件危机。软件组件仍然由手工制造，并且它们之间存在着许多相互关系和依赖。开发人员难以将这些相互关系文档化，以便系统其他部分的开发人员能够理解相互的期望和限制。维护人员也难以理解这些复杂的、缺乏文

档的相互关系。

于是，业界就去寻找一种可以减轻这种相互连接的影响的开发方法。目前正在出现开发方法学上的转变，这是从让我们可以更快和更容易地编写软件的方法到支持编写可理解软件的方法的转变。这并非一种悖论，这是将开发转向程序质量的一种趋势。

1.4 解决方案3：设计一种复杂而完善的语言

早期的程序设计语言，例如FORTRAN、COBOL、APL、Basic甚至是C等，都是为了更方便地编写出代码而设计的。这些语言都相对地小、简洁和易学。编写代码时不赞成拐弯抹角，而简洁的编程表达方式被认为是编程中最好的技巧。

目前，在编程语言的设计上有一个很明显的变化。现代的语言，例如Ada、C++和Java等的设计方法与早期的语言刚好相反。这些语言十分庞大，难以学会。用这些语言编写的程序不可避免地要比用传统语言编写的类似的程序要长得多。程序员要负责定义、声明以及对代码元素进行描述。

这种完整性保证了代码的一致性。如果程序员在程序不同的部分使用了不一致的代码，编译程序就会发现它们，并且要求程序员消除这些不一致的地方。对于传统的语言，编译程序会认为不一致性是程序员为了达到某个目的而故意引人的。设计和编写语言编译程序的人员则会辩解“我们不想对程序员的工作做猜测”。使用这些语言的程序通常要进行复杂的运行测试，但测试的结果仍然无法发现一些潜在的错误。现代的语言将不一致性看做是语法错误，于是要求在程序运行之前，程序员就已经全部消除了这些不一致性的错误。这是一个十分重要的优点，但却使编程变得更为困难。

这种完整性的另一个优点是程序员可以在代码中更好地表达自己的意图。如果使用传统的语言，维护人员经常要去猜测设计人员在代码中表达的含义。为了帮助读者理解，就需要在代码中加入详细的注释，但是设计人员通常会因为缺乏时间（或技巧）而无法进行充分的注释。现代语言使得代码设计人员的代码可以更好地自我文档化。“众多”的声明减少了在代码中加注释的需求，并使维护人员能更好地理解代码的含义。这是业界的一个新的发展趋势，我们将会看到支持这种方法的一些实际的例子。

这些现代语言都比较大而且复杂；当然，由于它们都太大和太复杂，所以管理人员、科学家或者工程师都会觉得难以掌握。这些语言是专为受过专业训练的程序员而设计的，这里的专业训练是指学会如何将整个系统划分为没有太多相互关系的相互协作的部分（也指开发人员之间不需要有太多的共享知识）。在传统的语言中，模块的基本单元是函数。在代码中无法显示某个函数与其他函数之间的逻辑关系的紧密程度。虽然，新的语言也使用函数作为模块化的单元，但是它还给程序员提供了将函数聚集在一起的手段。在Ada语言中，将这种聚集称为包。Ada包可以包含数据，包的函数可以对某个数据进行操作，但在Ada程序中只对那个数据的一个实例进行操作。在C++和Java语言中有了更进一步的发展：它们的聚集单位是类，在类中，将函数与数据结合在一起，使得程序员可以使用任意数量的数据实例，即对象。

然而，使用现代的程序设计语言本身并没有任何优点。使用这些语言编写的程序可能会与使用传统的语言编写的程序一样糟糕：程序的各个部分之间同样存在许多的链接，同样存在需要注释的意义含糊的代码，于是维护人员的精力要分散到不同层次的计算上。于是，一种面向对象的方法就应运而生了。下一节，将会介绍面向对象方法的优点。

1.5　面向对象方法的含义和优点

每个人（几乎是每个人）都对面向对象方法感到兴奋。几乎所有的人都知道这种方法要比以前的任何一种方法都好（尽管他们不知道原因何在）。对面向对象方法不觉得兴奋的人并不是真正怀疑此方法的有效性。他们只是怀疑为实现这种改变而付出的努力是否值得：要付出费用培训用户和开发人员；要努力去制定新的标准、指南和文档；要推迟项目的进度以便学习新的语言；要用新的技术改正相关的错误。

确实要冒很大的风险，但是也会有很大的回报。面向对象方法的最重大的进步来自于人们广泛地接受和使用支持对象的语言，这里C++所起的作用无疑是最大的。

面向对象方法是否只是一个一时流行的词语呢？在适当的时候，它是否会被其他的方法所取代呢？它是否具备了真正的优点呢？是否还存在一些缺点或缺陷呢？

可以坦白地说，没有任何理由使用面向对象方法（以及使用C++的类）编写小型的程序。面向对象方法只在编写大型和复杂的程序时的优点多于缺点。

程序的复杂性由两个方面决定：

- 应用本身的复杂性（指该应用程序能为用户做什么）。
- 程序实现上的复杂性（由设计人员所做的决策以及程序员实现程序的方法所决定）。

我们无法控制应用的复杂性，因为这是由程序的目标所规定的。我们同样不可以期望将来应用的复杂性会降低；如果有任何变化，那么随着硬件功能的不断增强，要求我们去完成更加复杂的任务，从而使复杂性增加。

我们应该控制的是复杂性的第二个因素。每当我们决定将工作的某一个部分放在程序的某个单元内实现，而将另一部分的工作放在程序的另一个单元内实现时，就意味着增加了单元之间进行合作、协调和通信的额外复杂性。如果我们将本来应该放在一起的动作分开到程序的不同单元中，就会有很大的风险。这样会大大增加程序的复杂性。

人们为什么会将应该放在一起的动作分开呢？没有人会故意这么做。然而，通常很难区分出什么动作应该放在一起，而什么动作应该分开实现。为了能够做到这一点，就要学会如何评价设计的质量，以及更深的内容，例如什么是设计？

1.5.1　设计人员的工作

大多数软件业人员认为，设计工作包括了确定程序应该完成的工作、它应该为用户实现的功能、它要生成的数据、它进行工作所需要的数据。另外还有的工作是：确定为了完成工作需使用什么样的算法、用户界面如何、要求具备什么样的性能和可靠性，等等。但这并不是设计，这是分析。

当我们了解了程序应该为用户实现的功能、程序的输入和输出数据、数据转换算法、用户界面等内容后，接下来的工作才是设计。通常，设计是以下的一组决策活动：

- 一个程序由哪些单元组成。在设计一个软件时，要确定一个程序所包含的函数、类、文件以及其他的单元。
- 这些单元相互之间的关系如何（谁使用谁）。这一步要决定哪个单元要调用另一个单元的服务，以及为了实现这些服务要交换什么数据。
- 每个单元的责任是什么？在做出这个决策时，最易于将本该放在一起的动作分开放到不同的单元中。但这个结论对于实际的设计来讲还是太理论化了。在实际的设计中，需要

掌握一些进行模块划分的设计技术。

请仔细思考一下，这不仅对软件设计适用，还对其他创造性的活动适用。就像作曲家作曲、编写一本书、给朋友写封信或者创作一幅油画一样，在完成这些事情的时候，都要决定最终的产品应该由哪几部分组成，这些部分的相互关系如何，以及每一个部分在实现共同目标中所起的作用是什么等。工作的复杂程度越高，设计对工作质量所起的作用就越大。完成一个简单的电子邮件消息并不需要详细设计，但编写用户使用手册就需要非常详细地设计。

结构化设计使用函数作为模块化的单位。设计人员首先要明确程序要完成的功能，然后将这些功能细分为更小的子功能，再细分为步骤、子步骤，直到它们足够小为止。最后，将每个子功能实现为一个具有单独功能的独立函数。

在以数据为中心的设计中，划分模块的方法是使每一个模块专门负责某个特定的输入或输出数据的处理。设计人员首先要明确说明用作程序输入的数据以及程序输出的数据。然后设计人员不断地将复杂的数据细分为较小的子数据，直到所需的可以实现将输入数据转换为输出数据的处理过程足够小为止。最后，将每一个数据转换实现为一个具有单独功能的独立的函数。

对于不同类型的应用，存在着许多上述技术的变种，这些应用包括数据库应用、交互式应用、实时处理，等等。

所有的高级程序设计语言都提供了函数、过程、子程序或者其他类似的模块化单元（例如COBOL的段）和支持这些模块化的设计技术。这些方法都很有用，但却不能消除设计复杂性的问题：由于模块是通过数据链接起来的，因此存在相当多的程序模块之间的相互联系。对数据的引用使得单元代码的意义模糊不清。设计人员（以及维护人员）需要考虑太多的因素，于是容易出现难以发现和改正的错误。

软件设计人员运用一些准则尽可能降低代码中各个部分之间的相互依赖和复杂性。传统的软件质量准则是内聚性及耦合度。当代的面向对象技术所使用的质量准则是信息隐藏和封装。

1.5.2 设计质量：内聚性

内聚性描述了设计人员放在同一个模块中的各个步骤之间的相关程度。如果一个函数只通过一个计算对象完成一个任务，或者通过若干个步骤共同地实现了某个目标，那么就称该函数具有很好的内聚性。如果一个函数通过一个对象完成了若干个不相关的任务，或者通过若干个对象完成若干个任务，就称该函数具有差的、低的或弱的内聚性。

具有高内聚性的函数是容易命名的，通常可以使用一个动词短语来命名：用一个动词表示动作、用一个名词表示动作的对象。例如：`insertItem`，`findAccount`。对于低内聚性的函数，就要使用几个动词或者名词，例如`findOrInsertItem`（当然，除非我们要掩饰设计人员的失误，即调用函数`findItem`的作用是在某个集合中查找某个项目或者当不存在时插入该项目）。

解决低内聚性的方法就是重新设计。重新设计意味着要修改功能划分以及各个功能组件之间的关系。对于存在低内聚性的情形，要将具有低内聚性的函数划分为若干个具有较高内聚性的函数。

这种方法通常是有效的，但是要注意不能走极端，即划分出太多很小的函数。否则的话，设计人员和维护人员就要花费很多时间去管理许多事情（包括函数的名字以及它们之

间的接口）。

这就是为何不能单独使用内聚性来评价设计的原因，它不是一个很强的准则。我们还需要其他的设计准则作为内聚性的补充。但是在评价设计时还是要先考虑其内聚性。

1.5.3 设计质量：耦合度

第二个设计准则是耦合度，它描述了一个函数（称为服务器，即被调用函数）与其调用函数（称为服务器的客户）之间的接口。客户为服务器函数提供输入数据值。例如，函数processTransaction（一个客户）调用了函数findItem（一个服务器），并将项ID以及出错信息作为输入数据传送给函数findItem。服务器根据给定的正确输入数据才能产生正确的结果（例如，找到所需的项，显示出合适的出错信息）。

客户依赖于服务器函数所产生的结果。例如，函数findItem会为其客户（processTransaction）产生是否找到所需项的标志，若找到，还建立所需项的索引。这就是服务器的输出。服务器输入输出元素的总数就是耦合度的量度。我们希望通过减少在函数接口中元素的数目使耦合度降到最小。

耦合度是一个比内聚性更强的设计准则。它对设计上的决策十分敏感，尤其是当设计人员将应该放在一起的操作分开时。这些设计决策会不可避免地加大模块之间的通信以及额外的耦合度。例如，事务出错处理应该在一个地方完成，而不应该在processTransaction和findItem两个函数之间分开完成。

当然，解决耦合度过大的方法还是重新设计：重新考虑由什么函数实现什么功能。如果数据操作的某个部分由一个函数完成，而其另一个部分由另一个函数完成，那么，设计人员就应该考虑是否应该将两者合并为同一个函数。这种做法可以减少设计的复杂性而且不必增加程序中函数的数量。例如，如果将出错处理从函数findItem移到函数processTransaction，就可以不必将出错信息传送给函数findItem以及再将标志返回给函数processTransaction。

注意　在第9章的具体代码例子中，还会详细地讨论内聚性和耦合度的问题。

至此，希望读者在阅读C++源代码的时候，能够从内聚性、耦合度以及将属于在一起的操作分开的观点来分析代码。

1.5.4 设计质量：将数据与函数绑定在一起

面向对象方法对提高设计质量有何帮助呢？请注意，改进软件质量并不意味着使代码看起来更优美，因为美观并不意味着减少了代码的复杂性。改进质量意味着使代码更独立，使代码更具自我文档化，以及使设计人员的意图更容易让人理解。

面向对象方法的基本思想是将数据和操作绑定在一起。我们将会用大量时间讨论如何从语法上做到这一点。然而，最重要的是，在开始了解有关的语法之前，应该明白为何要这么做，以及为何使用这种语法是有用的。

为何将数据和操作绑定在一起会带来好处呢？在程序设计的功能函数方法中，其存在的问题是，“独立的”函数之间是通过数据连接起来的。例如，一个函数对某个变量设置值，而另外一个函数使用该值（findItem设置索引值，而processTransaction使用该索引）。这就造成了两个函数之间的相互依赖（以及可能更多的函数之间的相互依赖）。

一种解决该问题的方法是合并这两个函数。如果有效的话，就可以使用这种方法。但是这种方法未必总是有效的。通常，我们可能要通过不同的客户函数重复地调用服务器函数。这时消除服务器函数（findItem）是没有意义的。

另外，其他一些的函数也可能设置和使用同一个变量的值（可能在deleteItem，updateItem等函数中使用项索引）。对于一个小型的程序，当出现错误时，不难跟踪所有访问和修改变量的值的实例，以便找到出问题的原因。对于一个大型的程序，这就变得困难了，尤其是对于不完全理解设计人员意图的维护人员来说。即使是原来的设计人员，经过了若干个星期或若干个月后，通常都会感到难以完全理解程序，也难以找到使用了某个特定变量值的函数。

如果将访问和修改某个特定变量的一组函数都放在源代码的同一个地方，就好办多了。这样就可以帮助维护人员（以及回到程序的设计人员）理解程序员在当初编写程序时的真正意图。许多软件设计人员这么做的原因是他们理解代码的自我文档化特征的重要性。

然而，通常这是很难做到的。通常，为了容易找到函数，我们将函数按字母次序放在源代码中。即使在设计时根据它们要访问的变量成组放在一起，但也是很难确保所有相关函数真正成组地放在一起。程序员为了快速修正某个问题，会在程序的某个地方增加一个访问某个变量的函数，这时并不认为是语法错误。编译程序会接受该程序，运行的结果也是正确的。但将来的维护人员可能根本不知道有这样的另外一个函数存在。对于函数方法的程序设计来说，无法保证所有访问和修改某个特定数据的函数都存放在程序的同一个地方。

面向对象方法将数据与访问和修改这些数据的函数绑定在一起，从而解决了以上问题。C++将数据和操作合并在一个较大的单元中，该单元称为类。我们不将相关的事情分开，而是将它们合并，从而减轻了要记住程序中还有哪些相关部分的负担。我们还可以将数据私有化，以确保只有属于同一个类的函数才可以访问这些数据。因此，设计人员的意图可以显式地通过类描述的语法单元来表达。维护人员可以确信不会再有其他的函数访问或修改这些数据了。

图1-5表达了一个服务器对象与一个客户对象之间的相互关系。每个对象由数据、方法以

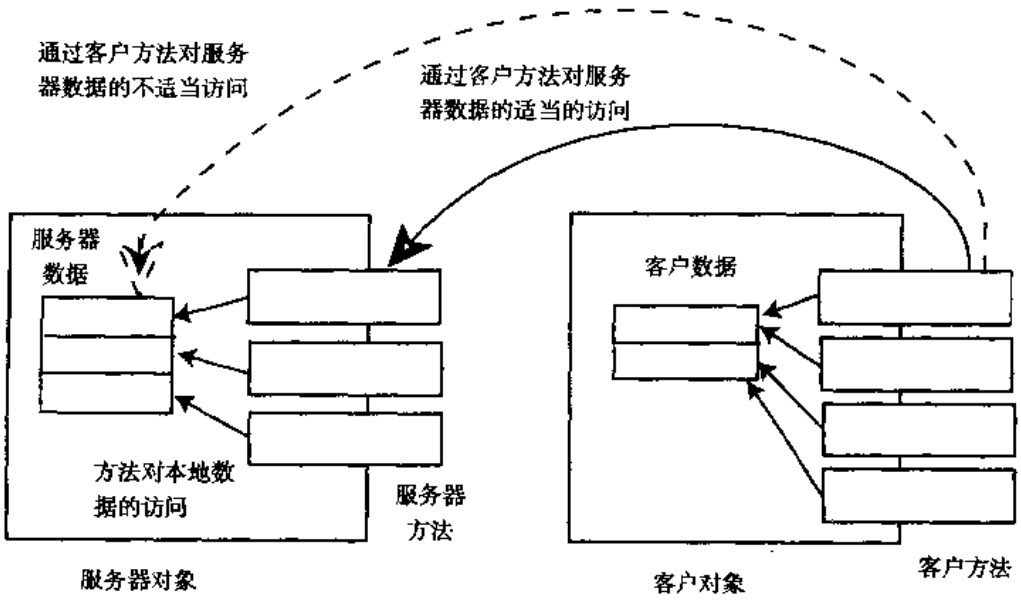


图1-5 服务器对象与客户对象之间的联系

及边界组成。在边界以内的所有东西都是私有的，是程序的其他部分不可使用的。边界以外的所有东西都是公共的，是程序的其他部分可以使用的。数据在边界以内，对于外界来说是不可见的。部分方法（函数）在边界以内，而部分在边界以外。在边界以外的部分是客户代码可见的方法接口。在边界以内的部分是客户不可见的隐藏的代码实现。

注意　这种方案是由Booch用Ada进行设计和编程时提出的。事实证明它对所有的面向对象编程和设计都十分有用。如果客户代码需要服务器的数据来进行工作，由于数据是私有的并且在服务器之外不可用，因此客户不能直接指出它所需要的服务器数据。解决的办法是，客户的方法调用服务器中能够访问所需服务器数据的方法。由于服务器方法是在服务器内部实现的，因此它访问私有的数据是毫无困难的。

1.5.5 设计质量：信息隐藏和封装

在确定应将什么数据与什么函数放在一起的时候，我们面临着要在众多的候选中进行选择的问题。有一些候选是明显不好的，而另外一些候选可能会更好。做出好的选择不是一件容易的事情。

封装的准则要求我们将数据与其操作合并起来，使得客户代码只需通过调用服务器函数，而不必显式地指出服务器数据成员的名称，就可以完成所需的工作。

这种方法的主要优点是维护人员可以很方便地从类描述中得知所有访问服务器数据的函数。

另一个优点是，由于客户代码含有较少的与数据有关的操纵，因而更容易实现自我文档化。例如，某个应用程序要对描述某个顾客的一组变量赋值，这组变量包括名字、姓氏、中名首字母、街道地址、城市、州、邮政编码、社会保险号，等等，总共有16个值。如果用传统的方法编写客户代码，设计人员就需要编写16条赋值语句。而现在，维护人员必须决定：

- 是否要对顾客描述中的所有组件赋值？
- 是否只是对顾客描述中的组件赋值，是否还有其他的数据也要处理？

要回答这两个问题是要花一定的时间和精力的，而其难度又与问题的复杂性有关。

如果使用面向对象方法编写上述的代码，那么数据是私有的，客户代码不能访问那些描述顾客的变量名称，例如姓名、地址等，只能调用访问函数`setCustomerData`，这样就可以将设计人员的意图清晰地传达给维护人员。

信息隐藏的准则要求我们将数据与操作合并起来，并且将功能划分到各个操作中，使得客户代码与数据设计之间可以相互独立。

例如，我们不必检查州、邮政编码等数据的正确性，只需检查它们是否一致。这时，客户代码可以将这项工作交给服务器代码去完成。当州的邮政编码要修改，或者某个城市要独立出来变成州时，只需要修改服务器代码，而客户代码不必做任何修改。如果我们将上述工作交给客户代码去完成，那么就要修改每一个使用到邮政编码的客户代码。

面向对象方法与传统的方法相比，最大的优越性就在于维护时只需做很少量的修改工作。假如我们要将5位的邮政编码转换为9位。如果使用传统的方法，就需要检查所有的客户代码，因为顾客的数据可能在代码中任意的一个地方处理。如果在某个地方遗漏了邮政编码的升位，那么，由于这并不是语法错误，在编译阶段是无法发现的，于是只能通过回归测试才会发现这个错误。

使用面向对象的方法去完成上述的邮政编码升位工作时，只需修改处理顾客邮政编码的函数，这对所有调用了该函数的客户代码都没有任何影响。所以，封装和信息隐藏的主要优点是：

- 通过将数据和函数绑定在类描述中，可以明确地表明什么函数访问了什么特定的数据。
- 从函数的名字就可以判断客户代码的含义，而不必理解大量的低层计算和赋值。
- 当数据表示发生改变时，该类的访问函数也要修改，但是只涉及到小范围的客户代码的修改。

1.5.6 设计问题：命名冲突

一个尽管不是最关键但还是相当重要的问题就是大型程序中的命名冲突问题。在一个C++程序中函数的名字应该是惟一的。假如某个程序员选择了findItem为函数命名，那么其他程序员就不能用同样的名字为其他函数命名。初看起来，这只是一个很简单的问题。想用同样的名字命名的人就会另外起一个稍作修改的名字，例如findInventoryItem或者findAccountingItem。

对于许多程序开发人员来说，保证命名的惟一性是一件令人十分烦恼的事情。问题不在于如何得到一个新的名字，而在于程序开发人员之间会有大量的通信。

假设你是一个由20个人组成的开发小组中的一员，并且将所负责编写的函数命名为findItem。再假设开发小组中有另外三个程序开发人员在他们编写的客户代码中要调用该函数。那么，如果使用面向对象方法进行开发的话，开发小组中只需要这三位程序开发人员知道你负责编写的函数命名为findItem，而其他程序开发人员没有必要知道这个情况，他们可以集中精力做其他的开发工作。

如果使用传统的开发方法进行开发的话，小组中20个人都要关注你的命名决定。当然，他们不需要一边工作一边了解你的命名决定，但是他们都必须知道你的命名结果，也必须知道其他人对其他函数的命名结果。请注意，你也必须了解其他所有函数的名字，尽管其中有许多是你根本不会用到的。许多部门为此制定了复杂的函数命名标准，并且花费了大量的资源来培训开发人员，以便让他们了解所制定的标准，强制他们执行这些标准，并且当标准发生变化的时候还要做出相应的管理。

这种做法很容易使人的精力无法集中到所进行的开发工作上。精力越是无法集中，就越容易出错。面向对象方法缓解了这一问题。它允许使用同名的函数，只要求这些函数属于不同的类。因此，只有那些确实要使用你所编写的函数的人员，才需要了解你对函数的命名，其他人员只需将精力集中在他们所负责的开发工作上。

1.5.7 设计问题：对象初始化

另外一个同样重要的问题是对象的初始化。在传统的方法中，用于计算的对象都要在客户代码中显式地进行初始化。例如，客户代码必须对顾客描述中的每一个组件显式地赋值。如果使用面向对象的方法，只需让客户代码调用setCustomerData即可完成初始化。如果没有调用该函数，这不会是一个语法错误，而是一个在运行时才会发现的语义错误。如果对顾客的处理要求使用到系统资源（文件、动态内存等）的话，这些资源也必须在客户代码中通过调用某个函数显式地被返回。

面向对象方法允许隐含地完成上述的操作。当客户代码通过传递初始化数据来建立对象时（以后我们将介绍实现这个操作的语法），不需要显式地调用初始化函数。计算结束后，分配给对象的资源也不需要通过客户代码中显式的操作就可以实现回收。

注意　这里只给出一般性的介绍，没有说明在C++中如何实现这些面向对象的特征。在后续章节有关的语法部分的学习中，读者应该时刻联系到此处对面向对象方法的特征的介绍，以免只见树木，不见森林。

1.5.8　对象的实质

在面向对象程序设计中，将程序设计为一组相互协作的对象而不是一组相互协作的函数。一个对象由数据和行为组成。作为一个程序员，你可能已经很熟悉有关数据的其他表达术语，例如数据域、数据成员或属性。我们将会经常提到一些对象行为的其他表达术语，例如函数、成员函数、方法或操作等。

数据表示了一个对象状态的特征。当相似的对象可以用相同的数据和操作描述时，我们可将对象的概念推广为类。一个类并不是一个对象。它描述了属于这一个类的所有对象的共同特性（数据和操作）。在程序运行时，并不是将类装入内存，而是将对象装入内存。某个类中的每一个对象都具有该类定义中的所有数据域。例如，每个InventoryItem都有一个i.d.编号、项目描述、库存量、购买价格、零售价格等等。我们将这些公共特性放在类InventoryItem的定义中。程序运行时，就会生成类InventoryItem的对象并为其数据域分配内存。这些对象可以相互独立地发生变化。如果某个对象的数据域的取值发生了变化，我们就称该对象的状态发生了变化。

同一个类中的所有对象都以相同的行为作为特征。对象的操作（函数、方法或操作）在类定义中与数据一起描述。同一个类中的每个对象可以完成同样的一组操作。这些操作作为程序中其他对象的代表而完成。通常，这些是对象数据上的操作。这些操作可以检索数据域的值，或者可以对数据域赋新值，或者比较数值，打印数值等。例如，某个库存项对象可以具有一个将零售价格设置为一个给定值的函数，或者具有一个将项的i.d.编号与某个给定编号进行比较的函数。

一个计算机程序可以具有多个属于同一个类型的对象。由于一个对象是一个类的实例，因此，使用对象一词来描述每一个对象的实例。有些人也使用对象一词来描述属于同一类型的一组对象。但是，人们一般还是使用“类”来描述属于同一类型的一组可能的对象实例。同一个类中的每个对象都具有其自身的数据域，但不同对象的相应数据域可以具有相同的名字。例如，两个库存项对象可以具有相同的（或不同的）零售价格取值；而不同的库存项对象可能具有不同的i.d.编号。同一个类中的所有对象可以完成同样的操作，也就是说，它们响应同样的函数调用。同一个类型的所有对象具有相同的特性（数据和操作）。我们调用一个改变对象的状态或者检索对象状态信息的对象函数时就说向对象发送一个消息。

这是一个十分重要的细节。在一个C++程序中，一个函数调用通常涉及两个对象。一个对象发出消息（调用函数），而另一个对象接收消息（有时称其为消息的目标）。我们称发出消息的对象为客户对象；称消息的目标为服务器对象；尽管这看起来像是客户-服务器计算机系统结构中的客户-服务器术语，但在此有着不同的含义。实际上，在面向对象程序设计中使用客户-服务器这些术语，比第一批客户-服务器系统流行时间要早得多。我们将会详细地讨论

C++程序中对象之间的客户-服务器关系。

我们将会在后面看到，在C++程序中，对象的语法形式与一般的变量——整数、字符以及浮点数的语法形式很相似。它们分配内存的方式也与一般的变量相似：它们分配在栈（stack）或堆（heap）中（对此以后将会给出详细的介绍）。C++类的语法形式是其他高级语言中将数据成员组合起来的结构或记录的语法形式的扩展。C++类中既包含了数据声明也包含了函数声明。

因此，当客户代码需要使用对象时，例如要进行项i.d.编号与给定数值之间的比较，或者要设置项零售价格的数值时，它并没有涉及到对象数据域的名称，而是调用该对象所提供的函数，并由这些函数为客户代码完成相应的工作：进行项i.d.编号的比较以及设置零售价格的值。

1.5.9 使用对象的优点

尽管听起来没有什么太大的差别，但是，客户代码使用对象的函数名还是使用对象的数据域名，这两者之间是有很大差别的。开发经验告诉我们，数据结构的设计比操作的设计更不稳定而且更容易发生变化。使用函数的名字而不是数据域的名字就可以使客户代码独立于服务器对象设计的一切可能变化。于是就提高了程序的可维护性，这是面向对象方法的一个重要目标。

另外，当客户代码调用一个服务器函数时，例如调用`compareID`时，客户代码设计人员的设计意图对维护人员来说是十分清晰的。如果客户代码要检索和操纵对象的`ID`，那么该代码的含义就要根据每一个基本操作的含义才能推断出来，而不能仅从函数的名字就可以得知。

总之，面向对象方法的目标与其他软件开发方法的目标是一致的：提高最终用户所希望得到的软件的质量（实现程序完整的功能和减少总的开发和维护费用）。

面向对象方法的支持者们希望，面向对象方法可以减少代码的复杂性。随着要处理的复杂性的减少，我们希望减少软件中可能发生的错误，并且可以增强开发和维护软件的生产能力。

软件复杂性的降低可以通过将程序划分为若干个相对独立的、可孤立理解的、对程序的其他部分引用较少的部分来实现。当我们将类作为程序的基本单位时，就有机会减少各个部分之间的相互联系。取而代之的是增强类中各个部分之间的相互联系；类的成员函数对同样的数据进行操作。这样做的效果很好，因为一个类通常由同一个人开发，正是开发人员之间的通信容易导致遗漏、不一致以及误解。减少类之间的相互依赖就可以减少开发各个类的开发人员之间的协调，以及减少出错的次数。

不像有相互联系的部分那样，相互独立的类容易在其他环境中重用；这就提高了开发系统的生产能力，还可以提高开发其他软件系统的生产能力。

有相互联系的部分要集中在一起进行研究，这是很耗时且容易出错的。而相互独立的类很容易理解，这就提高了程序维护时的生产能力。

面向对象技术并不是没有风险和代价的。程序开发人员、用户以及管理人员都要经过培训，而培训的费用是庞大的。使用面向对象方法的开发项目要比使用传统方法的项目花费更长的时间，尤其是在项目的初始阶段：分析和设计阶段。面向对象的程序要比传统的程序包含更多的代码（请不要担心，这里所指的是源代码的行数，并非指目标代码的大小，实际上目标代码的大小并不依赖于程序开发方法）。最重要的一点是，支持面向对象程序设计的语言

（特别是C++）相当复杂。因而使用面向对象方法会有以下的风险：无法实现上述的优点，面向对象程序可能比传统的程序更长、更复杂、更慢以及更难维护。所幸的是，本书不仅将会介绍如何使用C++，而且还会介绍应该避免出现什么问题。正确地使用这一强大的、令人兴奋的语言，可以帮助我们实现面向对象技术的承诺。

1.6 C++程序设计语言的特征

C++是C程序设计语言的超集。C语言本身是一些早期语言的几代后裔；它是在若干个互相冲突的目标下产生和实现的。这就是为何在C++中会含有一些不一致的并令人不满意的特征的原因。本节，将简要介绍C的主要特征，然后将会说明C++是如何“继承”这些特征来实现其目标的。

1.6.1 C语言的目标：性能、可读性、美观和可移植性

C的第一个目标是给软件开发人员提供一个面向性能的系统程序设计语言。因此C和C++不支持运行时的错误检查，这些错误可能会导致不正确的程序行为，但可以在程序调试期间由程序员发现。这就是为何C和C++会含有类似于汇编语言指令的低层次运算符的原因，这些运算符允许程序员控制计算机中寄存器、端口、标志位屏蔽码等资源。

注意　如果大家不知道什么是寄存器、端口、屏蔽码，请不要担心；这并不会妨碍我们掌握C++；只需要假定我们已经有过花费大量时间调试令人痛苦的汇编语言程序的经历即可。

C的第二个目标是为软件开发人员提供一个适于实现复杂算法和复杂数据结构的高级语言。这就是C和C++允许程序员使用循环、条件语句、函数以及过程的原因。也是为何C和C++支持处理不同的数据类型，包括数组、结构以及动态内存管理的原因。（如果对这些术语不熟悉，也不必担心，在使用这本书时，上述这些问题并不妨碍我们掌握C++。）这些特征支持实现代码的可读性和可维护性。

C的第三个目标是为了让软件开发人员写出优雅和美观的源代码。尽管没有给出“优雅”和“美观”的清晰的定义，因为这对不同的人来说会有不同的含义，但是一般人都承认，如果程序简洁、紧凑并且用少量结构良好的代码就能实现许多操作的话，就认为此程序是优雅和美观的。作为这种方法的结果，这种语言给予了程序员极大的编程“自由”，使他们不必担心在代码中会有许多的语法错误。今后将会对此进行更详细的讨论。

C的第四个目标是在源代码级上支持程序的可移植性。这正是C和C++的可执行目标代码不能在不同的操作系统或不同的硬件平台上运行的原因。但是，源代码可以不加修改地在不同的编译程序或不同的平台上编译，它可以不加修改地以同样的方式运行。

前面的三个目标已经很好地实现了，尽管它们之间有某种程度的冲突。用C语言开发的UNIX操作系统，已经逐渐变得很流行，可以在许多硬件平台，包括多用户环境（大型机、小型机以及PC服务器）以及单用户环境（PC）上实现。C语言还可以用来实现系统实用程序、数据库系统、字处理程序、电子表格、编辑程序以及许多应用程序。

可读性与程序表达简洁性之间的冲突仍未解决。看重程序可读性的程序开发人员就学习如何用C语言写出可读性好的代码。而注重程序表达简洁性的程序开发人员就学习如何用C语言编写简洁的代码，甚至有写出最晦涩难懂和最具表达力代码的比赛。

尽管同样实现了第四个目标——C代码的可移植性，但在实现上却有很大的保留。也就是说，语言本身是可移植的：如果在不同的操作系统或不同的硬件平台上，程序语句被编译通过的话，程序会以完全相同的方式运行。问题是在所有实际的C程序中，包含了比标准的C语言语句更多的内容：还包含了许多对库函数的调用。

设计C语言的隐含目标是要构造一个小型的语言。开始，它只有30个关键字。如果将它与COBOL或PL/I相比，会发现这个差别是令人难以置信的。因此，该语言很小。它没有指数操作，它不能比较或复制文本，它也不含输入和输出操作。然而，可以通过使用与编译程序一起提供的库函数来实现上述所有的（以及更多的）操作。语言的设计人员认为，应该由编译程序的供应商来决定应该使用什么库函数来比较或复制文本，以及来完成输入和输出等操作。

这并不是一个好主意。这不符合源代码可移植性的要求。如果不同的编译程序和不同的平台使用了不同的库函数，那么就不能在不修改对库函数调用的情况下将程序移植到不同的平台上。甚至在同一个平台上改用了不同的编译程序，程序也不能重新编译。同样，这种方法也与“程序员可移植性”的思想不一致。学习了如何使用一种库函数的程序员，如果要使用另外一种库函数的话，就要重新进行培训。

这不是一件小事，不同平台上的编译程序的供应商意识到了问题的重要性，因而为程序员提供了可以在不同机器上使用的不需做太多代码修改就可以运行的“标准的”库函数。不需做“太多”修改的含义是仍须做一些修改。由于缺乏权威机构进行标准化，因此已开发了若干个UNIX版本，并且在不同的机器和不同的操作系统上，来自不同编译程序供应商的库函数的行为会有所不同。

美国国家标准协会（ANSI）作为标准化的先锋，为了提高C语言的可移植性，在1983~1989年期间制定了ANSI C。语言的ANSI版本也结合了一些新的思想，但由于它具有向后兼容性，因此以往的C语言代码可以在新的编译程序下重新编译。

目前，尽管C源代码大多数都是可移植的，但是问题仍然存在，因此在将程序移植到不同的机器或不同的操作系统上时，仍需要进行修改。C程序员的编程技巧在大多数情况下也是可以移植的，程序员只需进行少量的培训（但需要进行一定的培训），就可以从一个开发环境转换到另一个开发环境中继续工作。

C的设计人员也没有意识到引入不同的库函数的“问题所在”。为了获得灵活性，我们付出了在移植代码和重新培训程序员时不断增加的代价。在处理这些问题的过程中软件产业所积累的经验，成了Java设计人员更加注重强调统一标准的原因之一。Java语言更为严格，它将许多C的习惯用法看成是语法错误。在Java的设计中，与C的向后兼容性问题被放在一个相当次要的地位。显然，Java 的设计人员不希望再重蹈覆辙。

1.6.2 C++语言的目标：与C语言向后兼容的类

C++的一个设计目标就是通过支持面向对象的程序设计方法来增强C的功能。请注意“增强”一词的确切含义，C++语言百分之百地设计为与C语言向后兼容：每一个合法的C语言程序就是一个合法的C++程序。（实际上也有一些例外，但它们都不重要。）因此C++具有C的所有好的以及不好的设计特征（直到永远）。

与C相类似，C++是面向标识符的，并且是区分大小写字母的。编译程序将源代码拆分为单词组件而不理会它们在源代码中的位置，因此不必限定代码中各个元素的书写位置。（例如，在FORTRAN或COBOL中就要限定）。C++编译程序会忽略标记之间的所有空白的空格，因此，

程序员可以利用空格来调整源代码的格式，以提高可读性。区分大小写有助于避免命名冲突，但是如果程序员（或者维护人员）不重视（或者没有时间重视）诸如大小写等细节问题的话，就会出现错误。

与C相似，C++只有很少的数值型的内部数据类型，比其他的现代语言要少。为了增强对破坏的免疫力，其中一些基本的数据类型在不同的机器上具有不同的取值范围。程序员可以使用所谓的修饰符，以便将变量的合法取值范围改变为某个机器可接受的范围，这样做会使事情变得更加混乱。其隐含的目的是为了实现可移植性和可维护性。

为了弥补内部数据类型的不足，C++支持将数据类型聚集为复合类型，包括数组、结构、联合以及枚举等类型。数据聚集还可以进一步合并为其他的聚集。这一特征也是从C中借用过来的。

C++支持一组标准的流控制结构，包括语句的顺序执行以及函数调用、语句的重复执行以及语句块（for，while，do循环）、判定语句（if，switch结构）、跳转（break，continue 还有goto语句）。这组控制语句与C的一样，但在使用for循环时有一些不同。

与C类似，C++语言是一种块结构语言，未命名的代码块可以嵌入到任意深度，在内层块中定义的变量在外层块中是不可见的。这使得编写内层块的程序员可以对局部变量任意地命名，而不必担心它会与编写外层块的程序员所定义的名字相冲突（以及需要协调）。

另一方面，一个C（以及C++）函数（即一个命名块）不能嵌套在其他函数内，因此函数的名字在程序中必须是惟一的。这是一个严重的限制。它增大了在开发过程中程序员之间的协调压力并使得维护变得更加困难。C++通过引入类作用域，部分地更正了此问题。类方法（即在类中定义的函数）只要求在类中惟一，而不必在程序中惟一。

C++函数可以像C函数一样递归地调用。传统的语言不支持递归调用，因为递归算法代表了所有算法的一小部分。天真地使用递归会浪费执行期间的时间和空间。然而，一些使用递归显得特别有效的算法确实得益于递归，因此，在现代程序设计语言中，递归是一个标准的特征（脚本语言除外）。

与C完全一样，C++的函数可以放在一个文件或多个源文件中。这些文件可以独立地进行编译和调试，这就使得不同的程序员可以独立地完成项目的不同部分。已编译的目标文件可以稍后连接起来，以生成可执行的目标文件。这对于实现大型项目的劳动力分工十分重要。

与C很类似，C++是一个强类型语言：例如，在表达式中或者向函数传递参数时，使用一个并非所期望的类型值是一个错误。目前，这是设计编程语言时普遍遵守的一个原则。很多要在运行时才表现出来的数据类型错误可以在编译时检查出来。于是节省了测试和调试的时间。

比C更进一步的是，C++是一个弱类型语言（是的，它既是强类型语言，又是弱类型语言）。在表达式和函数调用中可以自动地进行数值类型之间的转换。这与现代语言设计有很大的差异，它容易使一些错误无法在编译时发现。另外，C++支持相关类之间的转换。一方面，这使得我们可以使用“多态”这一很好的编程技术；另一方面，这一特征阻止了编译程序发现因疏忽而导致的错误。

C++以三种目的继承了C对指针使用的支持：1) 从调用函数向被调用函数传递参数，2) 从计算机的堆中动态分配内存（以实现动态数据结构），3) 操纵数组及数组分量。所有使用指针的技术都容易出现错误，这些错误最难检查、局部化和更正。

与C很类似，C++是为了提高效率而提出的：数组越界既不在编译时检查，也不在运行时

检查。要由程序员来维护程序的完整性以及避免因非法使用下标而导致的内存破坏。这通常是C/C++程序错误的来源。

与C相类似，C++是为了编写简洁而紧凑的代码而设计的：它对标点符号以及运算符赋予了特定的含义，这些符号包括星号、加号、等号、花括号、中括号以及逗号等。这些符号在一个C++程序中可能代表了多种含义。它们的含义由所处的上下文决定，这使得学习和使用C++语言比学习和使用其他语言更难。

C++在C的基础上增加了一些新的特征。最重要的特征就是支持对象。C++将C的结构扩充为类，它们将数据和函数绑定为一个代码单元。类通过将数据表示限制在其边界内，使得数据成员在类的外部不可见，从而实现了信息隐藏。类通过提供由客户代码调用的访问函数（方法）来支持封装，使用类作用域来避免C++程序中的命名冲突。

类提供了用于设计的层次方法，使得高层次的类可以重用低层次的类。类复合以及类继承使得程序员可以实现现实世界中复杂的模型并且更容易操纵程序的组件。

该语言还有许多其他帮助设计人员通过代码本身、而不必通过注释就可以表达设计意图的特征。

然而，与C类似，C++语言是为一个有经验的程序员而设计的。编译程序不会试图去猜测程序员的意图，因为它假定程序员很清楚自己在做什么。（但实际上，对于正在做什么，我们并非总是很清楚的，不是吗？）但知道我们正在做什么是很重要的。如果程序员不注意，一个C++程序就会相当复杂并且令读者生畏，因而难以修改和维护。类型转换、指针操纵、数组处理以及参数传递等都是C++程序中常见错误的来源。

所幸的是，本书介绍软件工程中有价值的经验，将会帮助大家明确自己正在做什么，以及避免出现那些会导致不必要复杂性的错误。

1.7 小结

本章，我们讨论了解决软件危机问题的几种方法。使用面向对象语言似乎是最有效地避免预算超支、进度延期、质量下降的方法。然而，用面向对象语言进行程序设计，要比用传统的过程化语言进行程序设计更难。正确地使用面向对象语言，可以方便阅读程序，而不是便于编写程序。实际上，有这一点已经非常好了，毕竟，只需在录入时书写源代码一次，但是却要反复阅读代码多次，包括要对它进行调试、测试以及维护。

作为一种面向对象语言，C++将数据和函数绑定在一个新的语法单元——类中，类扩展了类型的概念。使用C++的类，是将程序编写为一组相互协作的对象，而不是一组相互协作的函数。使用类有助于实现模块化、有助于设计具有高内聚性及低耦合度的代码。类支持封装、类复合以及继承，因而有助于实现代码的重用和可维护性。使用类消除了命名冲突并使代码更容易理解。

学习如何正确地使用C++是十分重要的。不加区别地使用从C那里继承下来的一些特征，会使C++无法实现面向对象程序设计的优点。我们只是对这些特征做了必要的、粗略的初步介绍。在本书的后续部分，将会着重讨论如何最好地使用C++的特征，以及许多特定的技术细节。建议大家最好能经常回到本章重温相关的内容，以便不会因为低层次的语法细节而忘了主流的面向对象思想。

下一章将开始讨论正确使用C++的方法，将讨论基本的程序结构以及最重要的程序设计结构。

第2章 快速入门：C++简介

在更深入地进行探讨之前，本章首先简要介绍C++语言的基本编程结构。由于C++语言内容丰富，凡是“简要”的部分不会涉及多少语言内容，而确实表明最重要语言特征的部分又不在“简要”之列。我将力求作出合理的取舍。

但是，如果逐个地研究C++的每一个特征，就不可能给出一个完整的、将不同的概念组成一个统一整体的认识。许多特征是交织在一起的，不能单独地介绍，因此就需要作这一简要介绍。首先，本书将会介绍C++语言中最重要的概念和结构，使我们能够编写出第一个C++程序，为今后能更深入而全面地学习这些概念和技术做准备。

本书的程序是按照ISO/ANSI的标准C++来编写的。这个语言版本新增了一些特征，也对现有一些语法特征做了修改。市面上还有许多编译程序只实现了这个新语言的部分特征。目前也有许多不同的供应商以及许多不同的编译程序版本讨论了他们在实现标准C++细节上的不同。最终，新的版本将代替旧的版本。但是，业界还要长期地处理按照早期标准C++版本编写的代码。由于向后兼容性是设计C++的重要的目标之一，因此新的编译程序版本也会支持旧版本的代码，因此增加了新特征的标准C++版本不会认为旧特征是非法的。鉴于此，在本书中将不会明确地指出新的标准C++语法。如果有必要的话，将会引用以往的编码方式，使我们能够有信心处理过去的代码。

2.1 基本程序结构

程序2-1给出的源代码是我们在本书中见到的第一个C++程序。程序显示"Welcome to the C++ world!"（就像大多数介绍编程语言的书的第一个例子一样）。另外，除了具有"Hello World"这样典型程序的功能以外，程序还进行了一些简单的计算，将pi（3.1415926）的平方结果打印输出。

程序2-1 第一个C++程序

```
#include <iostream>                                    // preprocessor directive
#include <cmath>                                       // preprocessor directive
using namespace std;                                   // compiler directive
const double PI = 3.1415926;                           // definition of a constant
int main(void)                                         // function returns integer
{
  double x=PI, y=1, z;                                 // definitions of variables
  cout << "Welcome to the C++ world!" << endl;         // function call
  z = y + 1;                                           // assignment statement
  y = pow(x,z);                                        // function call
  cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl;
  return 0;                                            // return statement
}                                                      // end of the function block
```

如果觉得该程序还不清晰，请不必担心。到本章结束时，大家就会明白这里的每一个细

节（甚至明白得更多）。

与其他现代的语言一样，C++允许编写其形式与人的阅读习惯相同的源代码。C++编译程序将源代码转换为机器可以识别的目标代码。在程序执行时，机器语言的指令逐条执行，并产生结果。

大多数计算都是对存储在计算机内存的数值进行处理。在我们的用法中，可以将计算机内存看做是由含有数值的地址单元组成的数组。存储在这些地址单元的数值不能引用这些地址。这些地址只能用数字地址（在目标代码中）或者用符号名称（在源代码中）来引用。例如，第一个C++程序中，含有以下的语句：

```
z = y + 1;
```

它指示计算机取出存储在名为y的地址单元中的数值，把该值加 1（不改变地址单元y的内容），并将结果存放到标识为z的地址单元中。名为y和z的地址单元的真实地址在可执行代码中指定，而不是在源代码中指定。程序员为这些单元命名，但并不关心编译程序为每个名称分配了什么内存地址。

在实际的内存中，为整数、浮点数以及字符（文本）分配不同的位数以及字节数，并且它们的位模式在运行时有不同的处理方式。为了正确地生成可执行代码，编译程序必须理解程序员的意图。这就是为何在执行语句z = y + 1之前，必须告诉编译程序y和z是内存某地址单元的名称（而不是其他的含义，例如函数），以及这些名称对应的内存单元中所存放的数值属于double类型（C++中一种带有小数部分的数值类型）。

因此，程序员编写的大多数源代码要么定义了程序操纵的对象（这里是x, y, z，其他的是通过#include指令和#define指令指定的数据），要么表达了要使用这些对象执行什么操作（这里有加法、赋值以及向函数传递参数）。

C++程序的源代码可以是一个由文本编辑程序生成的普通文本文件，这些编辑程序可以是Unix上的Emacs或Vi、VMS上的Edt、PC或者Mac上的集成开发环境（Integrated Development Environment, IDE）。这里我们将它作为一个文件保存在硬盘上。

通常，可以为源代码给定一个合适的名字，但对所用的文件扩展名是有限制的。根据编译程序的规定，源文件必须以文件扩展名.cc、.cpp或者.cxx来保存。使用其他的扩展名也可以，但会不太方便。如果使用标准的扩展名，那么只需指定源文件的名字，开发工具就会自动为文件加上扩展名。允许使用非标准的扩展名，但必须显式地指定它们。

一个源文件可以定义若干个函数（第一个C++程序只有一个函数，其名字叫做main）。一个程序可以由若干个源文件组成（上述程序只有一个文件）。每个源文件必须被编译为对应的目标文件。大多数环境都要求在程序执行以前，要先将已编译的程序链接起来。（本章稍后将会对此进行更详细的介绍。）图2-1给出了第一个C++程序的执行结果。

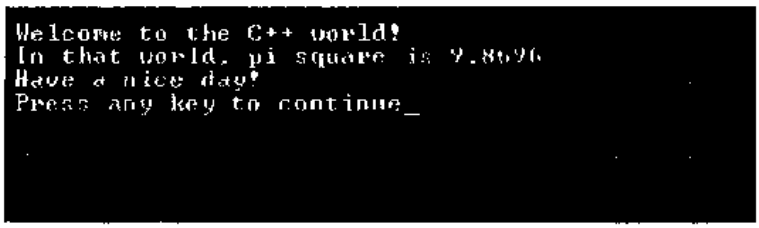


图2-1 由Microsoft编译程序产生的第一个C++程序的输出结果

这个输出结果是由Microsoft Visual C++编译程序的专业版6.0版本对程序编译执行后产生的。这是Microsoft Development Studio 的一个组成部分，它在同一个软件包中集成了若干个开发工具。程序由Development Studio调用。输出的最后一行是由编译程序而不是由该程序产生的。否则，将会在程序结束时马上清除屏幕的信息，于是用户就不能够观察到程序的输出结果了。Microsoft编译程序的早期版本没有增加这一信息，但也不会马上清除屏幕的信息，这项工作交给用户来完成。该程序也可以作为一个单独的应用程序在DOS提示符下运行。这时，就不会出现最后一行的信息。图2-2给出了在DOS提示符下该程序的运行结果。

```
C:\WINDOWS>echo off
Welcome to the C++ world!
In that world, pi square is 9.8696
Have a nice day!
C:\WINDOWS>
```

图2-2　在DOS命令提示符下第一个C++程序的输出结果

不同机器上的数值输出结果也有可能不同，这取决于对输出的数字位数的缺省设置。C++允许程序员显式指定不依赖于编译程序设置的输出格式，但这相当复杂，故不在此进行介绍。以后大家将会看到这样做的一些例子。

第一个C++程序给出了在所有的C++程序中可能包含的以下成分：

- 预处理程序指令。
- 注释。
- 声明和定义。
- 语句和表达式。
- 函数和函数调用。

下面几节，将详细地讨论每一种成分的使用。

2.2　预处理程序指令

在大多数其他语言中，编写的源文件就是编译程序在编译时所面对的源文件。但在C++中却并不如此。在将源代码转换为可执行程序的过程中，编译程序并不是第一个工具。处理源代码的第一个工具是预处理程序。它是什么呢？其实，它是C++从C那里继承下来的一个有趣的革新。它的目标是减少程序员在开发程序时编写的源代码的数量（或者在调试以及维护时阅读的源代码的数量）。

预处理程序处理源文件，并将处理结果传送给编译程序进行编译。预处理程序会忽略大多数的程序语句，并不做修改地将它们传送给编译程序。预处理程序只关注预处理程序指令（以及与它们相关的语句）。

预处理程序指令以'#'开头并占用一整行。不能在一个源文件行上写多于一条的指令。如果一行写不完一条指令，可以继续写在下一行，但前一行的末尾必须以一个特殊的转义符号'\'结束。符号'#'必须是一行的起始字符。什么是C++源代码的自由风格呢？正如第1章所述，可以使用一种自己（而不是编译程序）认为合适的格式编写C++代码，然而，预处理程序指令并不是C++/C语言的一个部分，预处理程序也不是编译程序的一个部分。

实际上，不使用预处理程序指令，即使是一个很简单的C++程序也无法编写。但是，从理

论上说，这些指令都不是语言的一个部分！实际上，是编译程序的供应商提供了预处理程序，但在理论上，编译程序和预处理程序没有任何联系。最近，编译程序的供应商放松了以上的限制：'#'号不一定是该行上的首字符，但它必须是第一个非空字符。

程序2-1使用了两条#include预处理程序指令。#include指令导致了直接的文本替换：预处理程序按照指令变量所给出的文件名取出完整的文件，并以该文件的内容直接替换该条预处理程序指令。这就可以将若干个源文件合并成一个源文件，然后将它作为一个整体进行编译。这种指令最常见的用法是，将描述源代码要用到的函数的函数头文件包含进来。

这些头文件的名字必须写在尖括号中，以便告知预处理程序要搜索在标准目录下的这些文件，编译程序在标准目录下存放了其头文件。例如，第一个程序中使用的#include指令指定了两个头文件。为了使用函数pow()，要用到第一个头文件，为了使用操作符<<以及对象cout，要用到第二个头文件。我们将在后面更详细地讨论函数、运算符以及对象。这正是C++复杂性的其中一个例子——如果不了解简单程序之外的内容，即使是对一个不使用无法理解的组件的简单程序，也不可能进行讨论。

```
#include <iostream>
#include <math>
using namespace std;
```

在第一个程序中，这一代码段的最后一行是指令using namespace。它不是预处理程序指令。它指示编译程序去识别由头文件引入的代码。这是一个新的语言特征。如果你使用一个早期的编译程序，编译程序会拒绝接受这三行。对于这样的编译程序，就不能使用using namespace指令。在早期的代码中，头文件的名字必须带有.h扩展名。因此，程序2-1中程序的开头三行必须用以下的两行代替：

```
#include <iostream.h>
#include <math.h>
```

预处理程序指令指示预处理程序在编译程序所在的目录下寻找include文件（计算浮点数幂的pow()函数、代表标准输出的cout对象以及在显示器上显示值的<<运算符）。

可以用其他的头文件来描述不属于标准库中的函数。这些函数通常由该项目的程序员编写。当在#include指令中使用这些文件的名字时，必须用双引号括起它们。例如：

```
#include "c:\work\mydef.h"
```

这条指令指示预处理程序将位于c:\work目录下的mydef.h文件的内容复制到源文件中。本例在文件名中使用的是绝对路径名。这样做对于源文件移到某一目录下，而头文件依然在另一目录的情况是方便的，这时指令不必进行修改。然而，通常是某个项目的整个目录树会移到另一个目录下，当头文件所在的位置改变了以后，使用该头文件的源文件必须随之修改。为了避免这种情况，程序员在#include指令中使用相对路径名。

#include指令在预处理程序处理之后，就会从源文件消失，并且不会传送到编译程序。

#include指令是十分重要的，没有它们程序就不能通过编译。然而，它们又不太复杂：程序员只需要知道什么函数要用到什么头文件，而编译程序的帮助功能可以帮助做到这一点。

程序2-1中的常量定义将符号名为PI的变量赋初值为3.1415926。（通常，为了与其数值在程序执行期间可以改变的变量相区别，符号常量使用大写。）接着，编译程序会处理下一行的代码：

```
double x=PI, y=1, z;          // definitions of variables
```

此可执行代码会将存放在地址PI的值复制到地址单元x中。另外一个使用常量PI的方法是使用#detine指令。#define指令也是用于实现文本替换的：它的第一个参数指出了被替换的文本，第二个参数指出了用作替换的文本。当预处理程序在后续的源代码中遇到了与第一个指令参数相对应的符号时，就会用第二个指令参数替换该符号。例如，在第一个C++程序中有：

```
#define  PI 3.1415926
```

该指令（现在的C++版本用const代替#define）指示预处理程序将PI的每一次出现用3.1415926来替换。当预处理程序处理代码：

```
{ double x=PI, y=1, z;              // definitions of variables
```

时，它会将下一行传送给编译程序：

```
{ double x=3.1415926, y=1, z;
```

请注意预处理程序会删除注释，使得编译程序不会处理到注释。（我们将在下一节讨论注释。）

可以用#define指令来定义宏，也就是插入到源代码中的一系列计算，而不是像上述例子那样的单一符号。从逻辑上来说，宏与函数在代码中的使用方式相同，都是用一个单一的名字来代表一组操作。宏的运行速度比函数要快，在C中普遍使用。在C++中使用的是内联函数而不是宏。因此，尽管在几年前，程序员必须知道如何正确地编写宏，但是现在并不打算详细地介绍宏。宏十分有趣，但是它也是导致错误和难以调试的原因之一。

其他重要的预处理程序指令控制条件计算。#ifdef指令可以将其后的代码包含其中，只要该指令中所用的符号已经定义。该指令的作用域由#endif指令限制。例如，如果已经定义了符号CPLUSPLUS，以下代码就会被包含和编译；否则，当程序作为一个C程序编译时，由于没有必要，预处理程序就会将代码隐藏起来使之不为编译程序所见。

```
#ifdef CPLUSPLUS
  . . . whatever is needed when the program is written in C++
#endif
```

请注意符号未必一定具有值；在#define指令中使用的符号，对于为了达到#ifdef指令的目的所定义的符号来说已经足够了。同样请注意符号的名字要用大写。尽管这一点不是必需的，但这是程序设计中通常的习惯。另外一种普遍的用法是使用小写，但开头的两个字符必须是下划线：

```
#define __cplusplus
  #ifdef __cplusplus
  . . . whatever is needed when the program is written in C++
#endif
```

另外一种指出#ifdef指令作用域的方法是使用#else指令。当跟在#ifdef指令之后的代码被包含到计算中时，跟在#else指令（直到找到#endif指令）之后的代码就不会包含到计算中，反之亦然。例如：

```
#define MT
  #ifdef MT
  #define NFILE  40
  #else
  #define NFILE  20
```

```
#endif
```

这段代码与我们在头文件中所见的内容相似；如果已经定义了符号MT，那么对文件数目的限制为40；如果从源文件中删除了该符号的定义，那么对文件数目的限制就是20。

#ifndef指令与#ifdef指令相反。只有在指令中所使用的符号没有定义的情况下，才会将其后的源代码内容（直到#else或者#endif指令为止的内容）包含进来。如果符号已经定义，就会跳过#ifndef之后的源代码；如果给出了#else指令，就将其后直到#endif指令为止的代码内容传递给编译程序。以下的例子也是来自于某个头文件：

```
#ifndef NULL
  #defne NULL   0
#endif
```

这是一个保证符号被定义，并且即使符号在多个文件中被重复使用，也只对符号定义一次的很常见的技术。如果在其他文件中又定义了该符号，有关的定义就会被忽略。

我们经常使用条件编译的预处理程序指令来实现程序的可移植性。如果应用程序要在几个不同的环境下进行工作，而且每个环境下的程序代码除了局部的程序段之外都基本相同，就可以将这些不同的程序段放在条件编译指令中。当系统从一个环境移植到另一个环境时，所需做的只是将定义某个符号的#define指令用定义另一个符号的#define指令来代替。

这看起来简单而有效，但实际上有一定的复杂性，预处理程序指令很容易被滥用。因此，在头文件中要限制使用预处理程序指令，一般情况下只允许使用#include指令。只有当你能自如地运用语言时，才可以使用其他的预处理程序指令。

2.3 注释

C++提供了两类注释：块注释以及行尾注释。块注释以一个双字符符号‘/*’开头，并以双字符符号‘*/’结束；行尾注释以双字符符号‘//’开头并以——是的，或许大家已经猜到了——行结束符（即：在源文件中的下一个换行字符）结束。双字符符号在C++中是很常见的，它们大部分源于C。使用双字符符号而不使用其他关键字表示操作并用于其他上下文中，是C的设计者为了设计一个只有30个关键字的语言而采取的方法之一。（可见C是一个很小型的语言。）

双字符符号的字符（C++中所有的双字符符号，不仅仅是注释）都必须连接输入书写。它们不能被空格符（或其他任何字符）分隔开。

两种注释中的文本逻辑上等价于空格符，因而对编译程序来说逻辑上是不可见的。实际上，注释对于编译程序来说不可见，这是由于预处理程序在源代码文本被编译前，已经将注释删除。以下是一个块注释的例子：

```
/* Comments are directed to a human, not to a compiler.
   Any symbol could be in a comment, including tabs, new
   lines, //, /*. We can format comments nicely, so that
   the structure of the text is clear to the reader. */
```

许多程序员将块注释作为函数或者算法的重要部分的前言。在这些注释中，描述了算法的目的、要处理的输入数据、作为计算结果的输出数据，以及为了完成任务所要调用的其他函数。通常也会记录程序修改的历史：第一作者和第一版本的日期、其他作者和修改日期、每一次修改的目的等。这些块注释的格式因人而异。当然，最重要的是坚持使用一种统一的

格式。更为重要的是要养成书写注释的习惯。有没有比书写注释更为重要的东西呢？当然有。这就是在修改了代码之后及时地修改注释。没有什么会比不正确的注释对维护造成的损坏更严重。

注意　在C语言中，只可以使用块注释。如果程序员想对某一单独的代码行进行注释，就要为这行书写与本书中的第一个C++程序类似的块注释。可能C的设计者并不认为程序员会因为在每一行的结束都要书写双字符符号‘*/’而感到麻烦。

程序2-2给出了第一个C++程序为早期编译程序而编写的版本：使用.h作为库头文件的扩展名，使用#define指令代替const，使用的是C类型的块注释。

程序2-2　带块注释的第一个C++程序

```
#include <iostream.h>                                  /* preprocessor directives */
#include <math.h>
#define  PI  3.1415926
int main(void)                                         /* function returns integer */
{
  double x=PI, y=1, z;                                 /* definitions of variables */
  cout << "Welcome to the C++ world!" << endl;         /* function call */
  z = y + 1;                                           /* assignment statement */
  y = pow(x,z);                                        /* function call */
  cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl;
  return 0;                                            /* return statement */
}                                                      /* end of the function block */
```

为了避免输入无用的字符，C++设计者为语言新增了行尾注释，其工作方式与块注释相同：所有位于双字符符号‘//’与下一个行结束符之间的内容对于编译程序来说，都是不可见的。

两种类型的C++注释之间有两个差别。第一个差别很明显：行尾注释不能跨行，而块注释允许跨行。因此语句中最开始使用的就是块注释。但这种差别几乎可以忽略（或者甚至变得无关），因为事实上行尾注释可以占用整行。

```
// Comments are directed to a human, not to a compiler.
// Any symbol could be in a comment, including tabs, new
// lines, //, /*. We can format comments nicely, so that
// the structure of the text is clear to the reader.
```

第二个差别更加微妙。行尾注释可以包含任何字符，包括其他的注释符号，其定界符是换行符（ASCII码为12）。块注释可以包含任何的字符，包括除了块尾注释符‘*/’之外的其他注释符号。也就是说，块注释不可以嵌套。预处理程序会忽略嵌套的开始符号‘/*’，这是由于它只是注释的一部分。当它遇到嵌套的闭括号‘*/’时，就会把它看做是注释的结束，于是会将其余的注释（包括第二个闭括号）传送给编译程序，这就会使编译程序困惑并产生错误信息。

```
/* Here, the second opening symbol /* is invisible */
   and the compiler thinks the last line is no comment */
```

这表明过去在程序员与编译程序（以及预处理程序）设计者之间存在着紧张关系，或者说，在我们所使用工具大小与它们的智能之间存在矛盾。C语言（以及后来的C++语言）偏向

于照顾工具设计者的利益。毕竟每个程序员都被告知不允许嵌套使用块注释。因此当错误发生时，程序员应该能够明确找出错误所在。

为何程序员希望使用嵌套的注释呢？通常，我们要试验代码的不同的版本，特别是当我们对代码的工作方式还不太有把握时。以第一个C++程序为例。如果我们想知道去掉中间三行后程序的工作情况，应该怎么做呢？最简单的方法就是将这三行代码变为块注释。如程序2-3所示。

程序2-3 代码块被注释而去掉的第一个C++程序

```
#include <iostream.h>                                    /* preprocessor directives */
#include <math.h>
#define  PI  3.1415926
int main(void)                                           /* function returns integer */
{
  double x=PI, y=1, z;                                   /* definitions of variables */
  cout << "Welcome to the C++ world!" << endl;           /* function call */
/* beginning of the block to be cut out
  z = y + 1;                                             /* assignment statement */
  y = pow(x,z);                                          /* function call */
  cout << "In that world, pi square is " << y << endl;
*/                                                       // end of the block to be cut out
  cout << "Have a nice day!" << endl;
  return 0;                                              /* return statement */
}                                                        /* end of the function block */
```

预处理程序能正确地将第一行（z = y + 1）作为注释去掉，但它将行尾的‘*/’符号看做是注释的结束。于是，预处理程序会不按照我们的原意，而是将第二行以及第三行传送给编译程序。最后还会将单个的块注释结束符‘*/’传送给编译程序，编译程序会因此而中断，并发出类似以下的出错信息（不同的编译程序会发出大不相同的编译错误信息）：

```
Compiling...

c:\data\ch02.cpp
c:\data\ch02.cpp(11) : warning C4138: '*/' found outside of comment
c:\data\ch02.cpp(11) : error C2059: syntax error : '/'

DEMO.EXE - 1 error(s), 1 warning(s)
```

这也是另外一个可以说明最好使用行尾注释符作为行注释的理由。如果在本例中改用行尾注释符的话，块注释就可以正确地工作了。当然，也可以使用前一节中描述的条件编译，但它们比块注释更复杂，更容易造成命名冲突，因条件编译主要用于将最终的程序从一种环境移植到另一种环境，而不是用于编写程序的实验过程。

还需对注释作进一步说明的是，在C++中，字符串表示的是用双引号括起的字符序列。在双引号内部，将注释符看做是解释性的文字，而不是注释定界符。也就是说，在双引号括起的字符串中，注释并不起作用。请看下面的语句：

```
cout << "Hello /* there */ world" << endl;
```

这个语句并不会输出Hello world，而会输出Hello /* there */world。

警告 在双引号之间的字符串中的块注释并不起注释的作用。如果想删除字符串中的文字，必须将它们删除，而不能仅仅对要删除的文字加注释。

提示　空行可以用来提高可读性，即用于区分逻辑上不同的代码段。但不要过度使用空行，因为这样会导致代码纵向的过度分散。

2.4 声明和定义

当程序员设计计算的逻辑流时，某一步计算的结果通常作为另一步计算的数据。因此，这些结果必须存储在存储器中，以便有必要时可以再读取出来。在第一个C++程序中，将y的值加1，其结果作为调用函数pow的第二个参数（它自乘第一个参数，以第二个参数为幂指数）。为了将数值存储在内存中以便将来使用时能迅速访问，这个值应该在计算机内存中有其物理地址。由于我们不希望在源代码中使用物理地址，因此程序员在程序中使用的是符号名字。

在程序2-1和程序2-2中，数值的和存储在地址y中，而1存储在地址z中。程序员并不关心如何将地址与名字联系起来，这是由编译程序的设计者解决的问题。程序员的任务是决定应该将什么样的数值存储在存储器中，以及用什么名字来代表这些数值。程序员给存储地址所起的名字在技术上被称为标识符。实际上，程序员不但要给变量设计对应的标识符，还要给常量、函数、数据类型以及标号等程序成分（以后将会详细介绍上述的程序组成）设计合适的标识符。

设计标识符的语法规则是很简单的：它们只能以字母或者下划线'_'开头，而不能以数字或者其他符号开头。组成标识符的其他字符可以是大写字母A~Z、小写字母a~z、数字0~9或者下划线。理论上，不限制标识符中总的字符个数。而实际上，如果两个标识符的前面31个字符（以前关于标识符长度的限制）都相同，编译程序将不能区分它们。如果觉得31位还不够用，只能考虑用名字缩写的方法。

除了可以使用下划线以外，不允许其他特殊符号（$，#等）出现在标识符命名中。也不允许嵌入空格符。

尽管以下划线开头的标识符是合法的，但是最好还是不要这样做，因为系统定义的标识符都是以下划线'_'或'__'开头的，于是有可能会引起不期望的命名冲突。只是在标识符的中间使用下划线来区分标识符的各个组成成分（例如sum_of_squares）。另外一种常用的技术是让标识符中的每个组成成分的开头字母用大写（例如SumOfSquares）。

良好的品味和谨慎的编程风格要求程序员使用便于记忆的名字给标识符命名。这里，便于记忆的含义是指标识符的名字能够在某种程度上，与它所代表的数值在程序中的使用目的联系起来。从这样的观点来看，在第一个C++程序中所使用的名字（x, y, z）并不是太好。只有当这些数值只被使用几次，只进行相当简单的计算并且不会使人产生误解时，才可以这样使用。名字PI就比较好，因为它表达了所代表的数值的含义（至少对于知道这个数的人来说）。

注意　C++是一种对大小写敏感的语言。这意味着如果程序使用两个只是字母大小写不同的标识符时，编译程序会认为它们是两个不同的标识符。例如，cnt、Cnt、CNT三个标识符在C++编译程序看来是三个不同的名字。这对于习惯用其他不区分大小写的语言来编程的程序员来说，是一个常见的出错原因。

如同上一节中所讲的，程序定义的常量通常使用大写，以便与其他在程序运行期间数值会发生改变的程序对象区分开来（例如第一个C++程序中的常量PI）。

C/C++的关键字是不可以用做程序员定义的标识符的保留字，它们都用小写。以下是按照

字母顺序排列的C/C++关键字：

```
auto  break  case  char  const  continue  default  do  double
else  enum  extern  float  for  goto  if  int  long
register  return  short  signed  sizeof  static
struct  switch  typedef  union  unsigned  void  volatile  while
```

以下是C++中而不是C中的关键字：

```
asm  bool  catch  class  const_cast  delete
dynamic_cast  explicit  export  false  friend  inline  mutable
namespace  new  operator  private  protected  public
reinterpret_cast  static_cast  template  this  throw
true  try  typeid  typename  using  virtual  wchar_t
```

在这里，并不要求大家马上记住所有这些关键字。我们将会在适当的时间对它们进行讨论。另外，如果将这些关键字用做变量的标识符，编译程序会指出不能这样做。因此，在此列出这些关键字的目的并不是要阻止大家使用这些关键字，而是要让大家明白为何在使用这些关键字作为标识符时，编译程序会发出出错信息。

在其他一些比较宽松的语言中，如果程序员想将某个值存放在内存中，他可以毫不困难地设计一个合适的标识符并将它用在赋值号的左边：sum_of_squares = 0。

但这在C++中是不允许的。如果你这么做，编译程序会告诉你sum_of_squares是一个未经说明的标识符，这在C++中是一个语法错误。在C++中，在某个标识符作为某个变量的名字之前，它必须先定义。

变量的名字代表了计算机内存中存放了某个类型值的地址。这些值在程序的运行期间可以改变。

这与第一个C++程序中的名字PI不同。在程序2-1中，将PI定义为常量，任何企图修改它的值的操作都是语法错误（例如，PI = 0）。在程序2-2中，它是一个由#define预处理程序指令定义的常量。当预处理程序用该常量的值代替了PI以后，它就变成一个不可以再发生变化的常量值。例如，语句PI = 0; 会转换为3.1415926 = 0; 这是一个语法错误。

内存中的变量存放已定义了类型的值，也就是说，程序员必须认可将要存储在这些变量中的值的类型。变量的类型决定了该类型的变量所允许的取值范围以及它所允许执行的操作。这种定义建立了标识符与其类型之间的联系，每个定义以一个分号结束。例如：

```
int num;
double sum_of_squares;
char letter;
```

第一个定义使用了关键字int，它表示将标识符num作为一个整型的变量来使用。它的大小为4个字节（32位），它的取值范围从-2147483648~+2147483647。（后面将会看到，类型的大小由机器决定。）

对整型所允许的操作有四个算术运算、求模运算（求除法的余数）、比较运算、左移或右移、逻辑操作以及加一和减一运算（将会在第3章中作详细的介绍）。

第二个定义使用了关键字double，它表示将标识符sum_of_squares作为双浮点型变量来使用。它的大小为8个字节，其绝对值（正或负）可达1.7976931348623158e+308(这里e+308代表了10的308次方，一个相当大的数)。这种类型上的操作包括四个算术运算、加一和减一运算以及比较运算。

第三个定义使用了关键字char，它说明了标识符letter是字符类型变量。它的大小为

一个字节，用于存储ASCII字符编码的字符。为了实现对字符的运算，C++将字符看做是小整型数。

整型值可用于计数以及数学运算。在每一台具体的机器中，整型上的运算是最快的，这就是为什么整型值常常用于范围和精度都足够的地方。浮点数带有小数部分，它们通常用于整数不能提供足够精度（或两百万兆的范围也不足够）的商业和科学计算中。

类型是功能程序设计和面向对象程序设计中的基本概念。C++程序处理的每一个值都必须具有特定的类型。如果不能正确地使用类型，编译程序就会发出语法错误的信息。这种情况通常发生在期望的类型与所使用的类型不相同的时候。如何使用正确的类型值通常是C++程序员考虑的焦点问题。

C++只有少数几个内部数据类型，也就是在语言中直接可用的数据类型。它们是整型（65）、浮点数类型（65.0）、以及字符类型（'a'）。

以上这些类型都是简单的（或标量）数据类型，也就是说这些类型中的数据不可以再分解为程序可以操纵的部分。例如，双浮点数类型由一个整数部分和一个小数部分组成（也有一个指数部分），但C++语言不允许程序员直接访问这些部分。只可以将此数作为一个整体来访问。因此，C++的类型是一个实现抽象的工具：它使程序员的精力集中在一个数值可以进行什么操作的问题上，而不必关注数值的各个组成部分是如何进行操纵的。

为了弥补基本类型的不足，C++允许使用一些其他大小、取值范围和精度的整型以及浮点型的变种。这并没有使情况发生很大的改变。更重要的是，C++提供了将各个简单的值组合为聚合，包括数组、结构以及类的技术。这些聚合类型的值是复合值，它们由一些组件组成；同时，C++支持对这些聚合的个别组件的访问技术。

当对变量定义进行运行时的考察（或实现）时，这些变量就会被分配相应的存储空间，不论它们是简单的还是聚合的数据类型变量。随后，我们就可以用这些变量来存储和检索相应声明类型的数值。这里没有什么特别之处，所有的现代强类型语言都是这样工作的。

一些程序员让每个定义占用源代码中单独的一行，使得每个定义都清晰可见。而另外的一些程序员却认为大量单独的短定义行会使变量的定义难以检查，于是，他们将若干个定义放在同一行上：

```
int num;  double sum_of_squares;
```

当变量属于同一个类型时，它们可以分别地定义，每个定义都包括了类型名并由分号结束，例如：

```
int a;  int b;  int c;
```

最好是将这些定义合并起来，即只使用类型名一次，属于该类型的变量名之间由逗号分隔，定义以分号结束。例如：

```
int a, b, c;                    // acceptable shorthand
```

也就是说，类型名（如上例中的int）的作用域包括了在类型名与下一个分号之间的所有的变量名：a, b, c三个变量都是整型变量。以上的两种定义方式是等价的，但是不要将它们的用法混淆起来。例如，以下的定义就是一个语法错误：

```
int a, b, int c;                // syntax error
```

另一方面，以下的定义是完全正确的：

```
int a, b;  int c;          // no syntax error
```

区别尽管很小，但却很重要。程序员应该能够意识到这两者之间的差别。

对于一个C++程序员来说，逗号与分号之间的差别是很重要的。请注意不要把两者混淆。

大多数程序员在函数块或文件的开头定义变量。这也是第一个C++程序的做法。在C语言中，这是惟一的一种定义变量的方法。然而在C++中，允许程序员在程序中靠近变量的初次使用的地方定义变量。程序2-4给出了一种比程序2-1更灵活的变量定义形式。

程序2-4 将变量定义放在代码中部的第一个C++程序

```
#include <iostream>                                    // preprocessor directive
#include <cmath>                                       // preprocessor directive
using namespace std;                                   // compiler directive
const double PI = 3.1415926;                           // definition of a constant
int main(void)                                         // function returns integer
{
  cout << "Welcome to the C++ world!" << endl;         // function call
  double y=1, z;                                       // definitions of variables
  z = y + 1;                                           // assignment statement
  double x=PI;                                         // definition of variable
  y = pow(x,z);                                        // function call
  cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl;
  return 0;                                            // return statement
}                                                      // end of the function block
```

对于程序的执行来说，在什么地方预先定义变量并没有什么区别。这个程序版本的输出与程序2-1的输出并没有什么不同。一般地，源代码中变量的定义与变量的使用之间的距离对于阅读者而言是不同的，特别是在变量只使用一、两次并且两次使用之间相隔不远的情况下。如果变量的第二次使用与第一次使用之间有间隔，而维护人员又要检查变量的定义，那么更方便的还是在函数的开头而不是在之中找到变量的定义。

另外一个应该熟悉的术语就是声明（declaration）。在其他的语言中，声明与定义是同义词。C++与C一样，认为两者之间有一些区别。定义使变量的名字对应了指定的类型，并为变量分配了相应的存储空间，而声明只是使变量的名字与类型之间建立了对应关系，因为变量的存储空间在其他地方分配。例如，这种情况发生在一个由多文件组成的程序中，在一个文件中定义了一个变量，而在另一个文件中使用了该变量。在使用这个变量的文件中，使用了关键字extern将变量定义为外部变量：

```
extern int count;
```

现在，我们可以在这个文件的源代码中使用变量count。在这个文件中所有对变量count的引用都会转换为在另一个文件中定义的变量count的地址。

另外一个定义和声明之间的区别是，变量在程序中只能进行惟一的一次定义，而声明的次数可以任意多次。例如，不允许进行以下形式的定义：

```
int a;  int a;                          // syntax error
```

另一方面，以下的声明是允许的：

```
extern int count;  extern int count;   // this is OK
```

这看起来并不明智，但有时候需要这样做。如果做错了，编译程序不会发出出错信息。

这里给出的只是一个基本的观点，将来会推广到函数和类型上。

警告　定义必须是惟一的，而声明可以重复多次。

当定义（或声明）了变量以后，程序就可以对这些变量进行操纵了。在程序使用这些变量之前，这些变量必须首先取得相应的值。如果没有这么做，就意味着使用了未赋初值的变量，这是程序中常见的错误之一。

有两种方法使变量获得初值：赋值语句以及初始化。在下面例子中，使用的是赋值语句（每个语句以分号结束）：

```
double x, y, z;
x = PI;  y = 1;
```

也许大家已经发现，在第一个C++程序中，使用的方法是对变量x和y进行初始化（z除外）：

```
double x = PI, y = 1, z;
```

结果是一样的，即变量x获得了值3.1415926536，变量y获得的值为1，而变量z仍未赋初值。从应用的观点来看，当我们处理简单类型的变量时，赋值的方法和初始化的方法之间的差别并不大。当开始要处理程序员定义的对象时，两种方法之间就有很大的差别。

变量只能通过定义而不能通过声明来初始化。例如，变量count只能在它所定义的文件（该文件为变量分配了存储空间）中进行初始化。在声明变量的文件中（将变量看做是一个外部变量），可以对变量进行任意的赋值和访问，但不可以对变量进行初始化。这种企图会认为是错误的，例如：

```
extern int count = 0;    // syntax error
```

这里只是对C++的数据类型这一问题作了介绍。第3章将会更详细地讨论C++的类型以及这些类型上的值可以进行的操作。

2.5　语句和表达式

语句是一个程序单元，它作为一个完整的逻辑单位执行，因此它的各个执行步骤对程序员来说都是不可见的。而这些步骤的细节不应引起程序员的注意（至少，不必每时每刻）。程序的语句是一种抽象工具：它使程序员的注意力集中在要做什么处理，而不是这些处理如何实现上。

例如，在上节中讨论的定义和声明都是语句。我们并不会对内存如何分配的细节感兴趣，例如，不会理会变量a的存放是接着变量b还是接着变量c，变量a的地址是否高于变量c以及一个字是否以高位字节开始等问题。我们只想确切地知道是否已经为三个整型变量分配了存储空间：

```
int a, b, c;
```

上节的赋值语句是第二种类型的语句。赋值的目标（接收数值的变量）写在赋值号的左边，赋值号的右边是将数值传给赋值目标的表达式。在第一个C++程序中，有以下的赋值语句：

```
z = y + 1;
```

当执行这条语句时，存储在变量y中的数值加1，并把加1的结果2存放到变量z对应的地址

单元中。这条语句的执行并没有影响到存储在变量y中的数值，只有当变量的名字位于赋值语句的左边作为赋值的目标时，它才会改变。

这里要重申，在程序2-1开始的定义中给出的是初始化，并非赋值。尽管语法形式是类似的，但C++对象调用了不同功能的函数：

```
double x=PI, y=1, z;    // x and y are initialized, not assigned
```

在赋值语句右边的表达式确实是表达式。它们由运算符和操作数组成。操作数可以是变量、数字或者更小的表达式。在第一个C++程序中，用来设置变量z的数值的表达式由操作数y和1组成。如果有必要，可以使用括号来构造复杂的表达式结构，例如：

```
z = (y + 1) * (y - 1);   // expression with subexpressions
```

这里，操作数y+1和y-1都是更小的表达式。每个表达式都会返回某一个类型的值，以便进一步用于赋值语句或者其他表达式中。

可以使用55个不同的C++运算符来构成表达式，包括算术运算符‘+’、‘-’、‘*’、‘/’、比较运算符‘<’、‘>’等等。要学习55个运算符数量实在是太多了。由于没有足够多的符号来表示这些运算符，C++使用了双符号的运算符。（例如，比较是否相等的运算符是‘==’）。C++将运算符组织成18个级别的优先次序。这也有很多内容要学。许多程序员喜欢用括号来指出运算的次序，而不是依赖于运算符的优先次序。下一章将会对此作更详细的讨论。

赋值号的左边部分与右边部分有一个很重要的区别。在C++的表达式中既可以使用变量的名字也可以使用数值。如果使用的是变量的名字（y），那么表达式所使用的并非该名字代表的地址，而是存储在该地址上的值。如果使用的是数值（1），那么表达式就直接使用这个数值，尽管这个数值也存放在某个特定的地址上。作为赋值的目标，只能使用变量的名字。尽管不是那么简单，但主要的事实是：一个字面值不能出现在赋值号的左边。例如，以下的式子看起来是一个等式，但却不是合法的C++代码：

```
1 = z - y;              // impossible in C++
```

第三种类型的语句是函数调用。在函数调用中指出了要执行的函数名字、要使用的参数（如果函数带参数）以及函数的返回值（如果函数有返回值）。

第一个C++程序包含了一个函数（main）。它使用了（调用了）函数pow。程序员以及技术人员有一种共同的习惯，就是在书写C++代码时，将函数名与所有其他的名字区分开来，这里的做法就是不论函数的参数有多少个，都在函数名的后面加上空括号，例如main()、pow()，与代码中的写法相似。

库函数pow()有两个参数：基数以及基数自乘的幂。返回的结果是以第一个参数的值为基数、以第二个参数的值为幂的自乘值。返回值可以用做一个表达式的成分，因此使用以下的语法来调用函数：

```
y = pow(x,z);
```

当第一个参数是3.1415926，而第二个参数是2时，返回的值就是9.869604。（与其他语言一样，非整型数的计算结果是一个近似值。）

当函数在程序执行期间被调用时，调用函数的执行就会暂停，而被调用函数就会执行其代码。当被调用函数终止或返回时，调用函数才会重新执行。如果调用函数又调用了其他函数，就会重复上述的过程。

解释C++的输入/输出库函数的调用比解释其他库函数的调用要复杂得多。它们使用了我们将在今后详细介绍的预定义的类、对象、以及重载运算符。在此先简单解释。我们使用了库对象cout（用于输出）和cin（用于输入）以及两类双符号运算符。插入运算符‘<<’与库对象cout一起使用，将要显示的变量的名字作为输出传送到计算机的屏幕上。用这种方法也可以输出数字型数值和位于双引号中的字符串。抽取运算符‘>>’与库对象cin一起使用，接受键盘输入的数据并存入到由操作数指定的变量中。

这看起来有点复杂，但是基本的输入/输出还是很简单的。由于每个输入或输出语句都要指出它自己的对象（cin或cout），它们是不会混用的。因此，输入和输出运算符‘>>’和‘<<’不能混用在同一个语句中。程序2-5给出了一个从键盘接收两个整型数据并显示它们的和的例子。

程序2-5 带有输入和输出语句的交互式程序

```
#include <iostream>
using namespace std;
int main(void)
{
  int a, b, c;                                    // definitions of variables
  cout << "Type two integers, press Enter ";
  cin >> a >> b;                                  // two function calls: extraction
  c = a + b;
  cout << "Their sum is " << c << endl;
  return 0;
}
```

图2-3给出了该程序的输出。



图2-3　带有交互式输入/输出程序的输出结果

每一次使用运算符<<和>>都意味着一个函数调用。endl库组件表达了一个所谓的操纵符。每一个输出元素都必须有自己的运算符，这些元素包括位于双引号中的字符串（以及在单引号中的字符）、数值（数字型数值）、变量或者是表达式。每个输入元素也应如此。用逗号或空格来分隔各个输入/输出组件是不正确的，例如，这是错误的：

```
cout << "Their sum is ", c  endl;          // typical errors: a comma, a space
```

双重损害使编译程序通常不能够正确地诊断出问题的来源。在出错信息中，通常会见到缺少分号、缺少参数、参数不匹配以及其他的一些有趣的问题。这里要提醒的是，我们不需要花太多的精力去解读编译程序给出的出错信息。只需要从出错信息中知道代码存在的问题，然后通过分析程序的逻辑去发现问题的所在。

警告　每个输入和输出的组件都必须有自己的>>运算符和<<运算符。不允许使用逗号或者空格来分隔这些组件。也不允许在同一条语句中混合进行输入和输出操作。

iostream库是强大而灵活的。然而，使用这个库来实现格式化输出的代码是很冗长的。

C++同样支持了另外一组“标准”的库函数：printf()和scanf()以及它们的变种。它们来源于早期的C，并且在早期的C和C++中都很常见。如果要使用它们，就应包含头文件stdio.h。使用这些函数来实现格式化输出比使用iostream的函数要更容易。然而stdio.h的函数会更容易出错。如今，iostream库比这些早期的函数更为流行，这些早期的函数已不再是“标准”的函数。不幸的是，我们恰好不能忘记这些stdio.h库而完全转向使用iostream库，因为stdio.h库函数经常用在Windows和GUI的程序设计以及字符串处理中。

与其他类型的语句类似，函数是一种抽象工具。在客户源代码中，函数指出了要做什么（就像第一个C++程序中的一样），但却不必描述具体的如何实现的细节。

我们已经讨论了三种最常见的语句：定义（和声明）、赋值以及函数调用。第四种类型的语句是类型定义。类型定义将组件合并为聚合类型，例如一个结构或者类，它们可以作为一个整体来操纵。我们将首先在2.6节“函数和函数调用”中简要地介绍类型定义，而书中的其他大部分的内容将会讨论类的编程问题。

最后一种语句是复合语句或语句块。它是一个由花括号括起的语句序列。复合语句可以和简单语句一样出现在程序的任何位置上。（包括出现在另外一个复合语句中。）例如，在第一个C++程序中，可以将最后的三个语句合并到一个语句块中，如程序2-6所示。

程序2-6 带嵌入语句块的第一个C++程序

```
#include <iostream>                          // preprocessor directives
#include <cmath>
using namespace std;
const double PI = 3.1415926;                 // definition of a constant
int main(void)
{
  double x=PI, y=1, z;                       // definitions of variables
  cout << "Welcome to the C++ world!" << endl;
  z = y + 1;
  y = pow(x,z);
  {                                          // start of statement block
    cout << 'In that world, pi square is " << y << endl;
    cout << 'Have a nice day!" << endl;
    return 0;
   }                                         // end of statement block
}                                            // end of the function block
```

这里的修改并没有带来太多的东西，程序的运行和以前一样。然而，在C++语言中，有许多语言结构（例如条件和循环结构）只具有容纳一条C++语句的位置，如果不能将所有这些计算逻辑都合并为一条语句，就会出现问题。使用语句块就解决了这个问题。

单字符的块定界符‘{’和‘}’必须成对出现，否则就会引起语法错误。它们表示了某个区域的开始和结束，即程序的一个结构化元素。可以观察到main()函数也是由花括号定界的，而源代码中的每一个函数也是如此。复合语句与函数体之间的区别在于：复合语句没有名字（它是没有名字的块），而函数是有名字的块。

C++程序中的语句是一个接着一个、自上而下地执行的。每个语句最好单独占用一行并适当地缩进，使得我们容易识别每一个控制结构。例如，在程序2-6的main()函数中，每个语句都相对于main()函数首部和预处理程序指令向右做了缩进，在main()函数中，位于嵌

套块中的语句又比其他语句向右进一步地缩进。

允许将几个语句放在同一行，如果它们是为了达到某个共同目标的连续步骤：

```
z = y + 1;  y = pow(x,z);       // two substeps of the algorithm
```

同一行上的语句是从左向右执行的。可以将多少条语句放在同一代码行上呢？其答案与阅读源代码时眼睛的最少移动问题有关。如果将每一条语句放在单独的一行，那么代码段就会变得很长。因此，眼睛就需要阅读长距离的代码，以致当我们读到一页的末尾时，我们可能已经无法记住此页开头部分的内容或者前两页中的内容。将若干条语句放在同一行就可以缓解此问题——因为我们一眼就可以看到若干条语句——但是眼睛却要在水平方向作较大量的移动，同时还会有一个危险，就是可能会遗漏了一行中间位于其他语句之间的某个动作。很大程度上，这是一个编程风格的问题。但无论如何，放在同一行上的语句都必须是为了实现同一个目标的。

在C++中，每个语句都必须以一个分号结束。在其他一些语言中，分号用于分隔语句，因此，语句序列中的最后一条语句就不必有分号。在C++中每个语句都必须以分号结束。但也并不总是这样。复合语句就不必以分号结束。请注意在第一个C++程序中，两个（用于未命名块以及main()函数）右括号之后都不必有分号。

实际上，分号可以将一个表达式转换为语句。例如，以下是一个合法的C++语句：

```
y + 1;
```

这个语句当然没有什么作用。其他的语言不允许这样使用，但它在C和C++中是合法的。为何要担心一个没有用的东西是否合法呢？程序员绝对不会写出这样的语句，对吗？不。这种语句并不像看起来那么无害。如果我们不小心写错了，例如，意外地删除了赋值的目标，编译程序并不会针对这项错误而为我们提供保障。本来应为语法错误的问题（在其他比较严格的语言中，这是一个错误）并没有被及时地发现，它成为一个运行时的错误，而且需要在调试程序时通过艰苦的努力才会发现。

控制流结构也可以看做是一种语句。它们改变了语句执行的顺序结构。有三种类型的控制流语句：

- 条件语句。
- 循环。
- 函数调用。

注意　本节只讨论控制流结构的一个小的集合。在第4章“C++控制流”中，将会进行更详细地讨论。

最简单的条件语句是if语句。它通常的形式是：

```
if (expression) statement_to_execute;
```

从语法上看，if语句是简单语句。当statement_to_execute语句是简单语句时，它以一个分号结束。如果是一个复合语句，就不需要在闭括号之后书写分号。条件表达式两边必须有括号，这是必需的。

如果表达式为真，就会执行条件表达式后面的statement_to_execute；如果表达式为假，就会跳过该语句。经常为了强调控制流而使用缩进。下面这一段代码用于检查气温是否在华氏冰点以上，若是，就显示信息；否则什么也不显示：

```
if (fahr > 32)                  // expression is commonly on a separate line
```

```
  cout << "Do not worry about starting your car" << endl;
```

第二种形式的条件语句有两个分支：当条件为真时，执行一个分支；当条件为假时，执行另一个分支。每个分支都由一个语句（用分号结束）或复合块（不带分号）组成。例如：

```
if (fahr > 32)                    // no "then" keyword in C++
  cout << "Do not worry about starting your car" << endl;
else
  cout << "Be careful in the morning" << endl;
```

C++中没有then关键字，它是隐含的。必须使用else关键字。请注意，缩进是为了强调控制流。

最简单的循环语句是while语句：

```
while (expression) statement_to_execute;
```

从语法上看，循环体是一个简单语句。因此简单语句statement_to_execute必须以分号结束。当循环体是复合语句时，在语句结束的闭括号之后就不用加分号。表达式两边的括号是必须的。

如果表达式为真，就会执行循环体statement_to_execute，然后再次测试循环表达式。如果为真，就继续执行循环体，然后又测试循环表达式。如果循环表达式变为假，就跳过循环体，并执行循环语句的下一条语句。

在程序2-7中，程序计算了8、9、10以及11的平方，并以表格的形式显示了这些数以及它们的平方（有关的输出在图2-4中）。它首先输出了表头和一个空行，然后使用了num作为循环变量。在循环之前，将num初始化为8；这个值首先用于第一次循环。在循环体中，将num加1；在循环条件中，检查num的值是否还是小于12，如果小于12，就继续执行循环体（位于括号中），并且num的值继续加1。循环继续执行，直到num的值变为12；当循环的条件变为假时，就跳过循环体，执行程序的最后一个语句。

程序2-7 一个带格式化输出的循环程序

```
#include <iostream>
#include <iomanip>
using namespace std;
int main (void)
{
  int num = 8, square;                          // num is initialized before loop
  cout << "Numbers Their Squares" << endl << endl;
  while (num < 12)                              // num is used as a loop variable
   { square = num * num;                        // num is used in the body
     cout << "   " << num << "      " << square << endl;
     num = num + 1;                             // it is modified at loop end
                 }                              // no ';' at end of the block
  cout << endl <<"Have a nice day." << endl;
  return 0;
}
```

当运算符<<使用对象cout将字符传送到屏幕时，它将字符依次地、不带空格地输出到屏幕上。即使它完成了将数值（例如num）从二进制形式转换为字符并开始转换另外一个数值（例如square）时，也不会插入空格符。这种无格式化的输出显然是不易读的。为了更快地实现“脏”输出，可以将空格插入到输出组件中。在程序2-7的循环中，cout语句就是这么

做的。

这种格式化的方法也是不能令人满意的。每一次不同的循环都会产生不同的字符个数，于是使各个列无法对齐，如图2-4所示。这个问题可以通过使用一个setw操纵符来解决，它用来指定下一个输出组件所占的输出位置的数目（输出宽度）。setw操纵符必须像其他输出组件一样与操纵符<<一起插入到输出流中。例如，如果插入了setw(4)，就可以为下一个输出组件分配4个输出位置。如果要对若干个组件进行格式化，每个组件之前都必须指定相应的setw操纵符，尽管每个组件的输出宽度都一样。

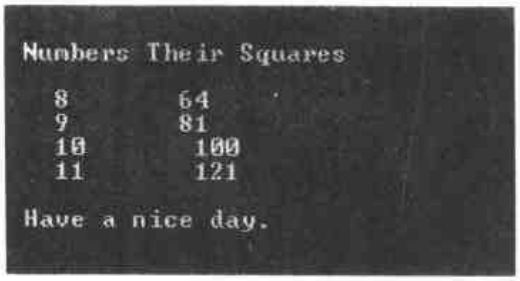


图2-4　循环计算各个数的平方后的输出结果

让我们将程序2-7中的循环cout语句用以下的语句代替：

```
cout << setw(4) << num << setw(10) << square << endl;
```

在这种工作方式下，程序中必须包含头文件iomanip（见程序2-7）。这个程序版本的输出见图2-5。对于数字型数值，输出的数据在指定的宽度内向右对齐，而对于字符串，则是向左对齐。如果输出的数据在指定的宽度内放不下，cout对象就会在屏幕上预留足够的位置，以便在右边继续输出其余的数据部分。输出数据不会因为超过了指定的宽度而截取掉。

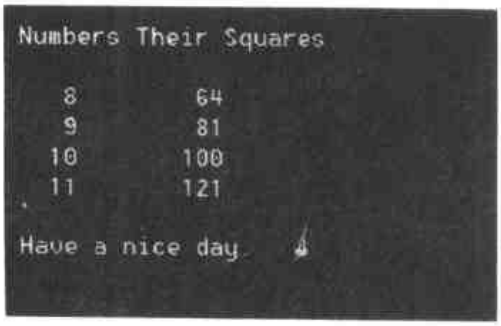


图2-5　为每个输出数值指定宽度的循环输出结果

必须保证循环设计中涉及到的各个元素以及描述的循环迭代是适当的。一个设计正确的while循环必须由以下的几部分组成：

- 在循环之前对循环变量的当前值进行初始化。
- 在循环体中使用循环变量的当前值。
- 在循环体中修改循环变量的当前值（通常是在循环体的最后）。

在程序2-7的例子中，变量num用作循环变量。它在定义时，即循环之前已初始化。在循环体中使用了num的值。在循环体的最后一个语句中对num的值加1。程序中利用了num的值来决定是否满足循环结束的条件。

2.6 函数和函数调用

函数模块化使得我们可以将软件的开发工作分配给若干个程序员共同完成。我们将各组函数放在不同的源文件中，指定每个程序员负责一个文件，并让这些程序员并行地工作。显然，一个程序员可以编写若干个函数或者若干个文件，但是几个程序员却不能同时编写同一个函数。如果函数太大，以致要若干个程序员一起进行编写的话，就应该将它分为若干个函数。

使用函数的优点还有：

- 调用函数的代码用函数调用表示（函数的名字应该能够反映操作的含义），这比低层次计算更具有可读性。
- 如果在源代码的不同位置上要完成相同的操作，源（和目标）代码的长度可以大大减小，在源（目标）代码中只需要重复长度较短的函数调用，而不必重复编写较长的低层次计算。
- 使用标准库（将工程专用的函数放进工程库中），提高了工程中以及工程之间源代码的可重用性。

将程序分为几个不同的函数会改变程序的结构，却不会改变程序的输出（如果各个函数都是正确的）。然而，对于不同的模块划分，程序的质量可能会有很大的不同：独立的函数能使程序更容易理解和维护。

让我们来讨论程序2-1的各种不同的可能实现方案。由于这个程序很小，这些例子都无法体现可读性、程序大小以及可重用性等方面的优点。然而，从这些例子中我们可以看到使用函数的语法和语义。

在第一个重新设计的例子中（见程序2-8），将问候用一个单独的函数displayInitialGreeting()来实现。这个函数被main()调用。因此，main()是该函数的客户，而函数本身是main()的服务器。

程序2-8 带有一个服务器函数的第一个C++程序

```
#include <iostream>
#include <cmath>
using namespace std;
const double  PI = 3.1415926;

void displayInitialGreeting()                          // function header
{
   cout << "Welcome to the C++ world!" << endl;        // its body
}                                                      // end of the function block

int main(void)
{
  double x=PI, y=1, z;
  displayInitialGreeting();                            // function call
  z = y + 1;
  y = pow(x,z);
//  cout << "In that world, pi square is " << y << endl;
//  cout << "Have a nice day!" << endl;
  return 0;
    }                                                  // end of the function block
```

当然，displayInitialGreeting()函数是一个相当笨的函数，由于它只有一个语

句。然而，即使是这么一个笨的函数，也足以说明在C++中使用函数需要在程序的三个元素之间进行协调：

- 函数头部。
- 函数体。
- 函数调用。

函数的头部指出了函数接口：返回值的类型、函数的名字、带有形式参数（如果有）类型和名字的参数表（位于括号中）。如果没有参数，那么位于括号中的参数表为空。如果函数没有返回值，那么返回值类型为void。

函数的名字描述了处理的含义，并且通常将描述动作的动词（display）与描述动作对象的名词（InitialGreeting）组合成函数的名字（displayInitialGreeting）。按照通常的编程习惯，我们将函数名的开头字母写成小写，而其他单词（如果有的话）中的第一个字母则用大写。

可以看到displayInitialGreeting()函数没有返回任何数值（它的函数返回值类型为void）。因此它不需要return语句。如果我们想要用，也可以使用，但不应返回任何数值。函数也没有任何参数（它的参数表为空）。但即使没有参数，也要在函数头部书写一对括号，并用关键字void指出没有形式参数：

```
void displayInitialGreeting(void)                    // void in parameter list
{
  cout << "Welcome to the C++ world!" << endl        // function body
  return;                                           // avoid unnecessary code
  }
```

函数体是由一对花括号括起的语句序列，它是一个语句块（一个复合语句）。每个语句都以分号结束，但块本身的结束不需要分号。如果有必要，可以在函数体内对要在函数中进行运算的变量进行定义和声明。每个函数体都有各自的名字空间（作用域）。这意味着在一个函数内定义的局部变量的名字并不会与在其他函数内定义的变量的名字产生冲突，因此函数的设计者不需要与其他函数的设计者协调局部变量的命名。

与C类似，在C++中函数定义不能嵌套，因此，函数名在程序中是全局变量，在整个程序范围内必须是惟一的。函数的设计者都必须与其他设计者协调函数的命名，不管他们是否需要调用这些函数。

displayInitialGreeting()函数并没有定义任何局部变量。为了强调函数体的界限，开花括号和闭花括号在源代码中都单独占用一行。有一些程序员认为这样会使源代码的纵向长度加大，对提高程序的可读性没有好处，因此，不让花括号单独占用一行，只是让函数之间由空行分隔：

```
void displayInitialGreeting()        // function header
{ cout << "Welcome to the C++ world!" << endl; }
                                     // function body
```

第三个使用函数的元素是函数调用，它是由函数名后面接着写在一对括号里的实际参数表组成的。如果没有参数，仍须保留一对括号。与在函数的头部中不一样，关键字void不能用在函数调用中。

```
displayInitialGreeting(void);        // incorrect function call
```

这也是C++从C那里继承下来的另一个令人费解的特征。对于C语言的设计者来说，设计的简单性从来没有被优先考虑。毕竟，是否需要花太多的时间去记住函数的头部可以使用void而函数调用中却不可以使用呢？确实不用。但语言的设计者没有想到的是，诸如此类的特征积累下来就会给程序员带来困惑。

实际上，一个真正的C语言专家知道如何回答这一问题，甚至可能认为这个问题很简单。但对C++而言却并非如此。作者曾给许多C++程序员讲过课，他们中的大多数都是很好的程序员，其中一些还是优秀的程序员，但作为讨论这一问题的参与者，他们很少能够对这一问题进行回答，除非是在获得了足够的鼓励和提示之后。对于这一问题，本书将在讨论了更多的相关内容之后给出答案。

当调用一个函数时，调用函数（客户函数）就会暂停其执行，并将控制权交给被调用的函数（服务器函数）。各语句将会顺序地执行。当执行到达服务器函数体的闭花括号时，服务器函数的执行就会结束，并将控制权交回给调用函数。（因此函数的终止又叫做返回。）随后，客户函数又重新执行。

下面来讨论带参数的函数。程序2-9给出了第一个C++程序的另外一个版本，其中有一个函数displayResults()，它实现了上一版本中局部块的功能。函数接收一个double类型的值作为参数，并在屏幕上显示该参数以及其他的用户信息。

程序2-9 有两个服务器函数的第一个C++程序

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

void displayInitialGreeting()                          // function header
{ cout << "Welcome to the C++ world!" << endl; }      // its body

void displayResults(double y)                          // function header
{ cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl; }                // its body

int main(void)
{
  double x=PI, y=1, z;
  displayInitialGreeting();                            // function call
  z = y + 1;
  y = pow(x,z);
  displayResults(y);                                   // another function call
  return 0;
     }
```

函数displatResults()并没有将任何数值返回给其调用函数（其返回值类型为void），但它的参数表非空。可以看到，参数的定义形式与变量的定义形式是相似的：程序员为参数命名并指定其类型。程序运行时，参数的定义结果与变量的定义结果相似：调用函数时，为参数分配相应类型的内存，当函数体中使用到参数名时，就会访问该名字对应的地址。（例如，在第一个cout语句中引用变量y。）用调用函数时的实际参数的值来对形式参数的值进行初始化。

函数调用时指出了函数的名字以及列在括号中的实际参数表。这里，表中只有一个参数

(它的名字与形式参数的名字相同，但这仅仅是巧合)。请注意实际参数是一个表达式，并不是一个变量的定义：只需在函数头部说明实际参数的类型，而不需要在函数调用中进行说明。以下的函数调用形式是不正确的：

```
displayResult(double y);          // error
```

下面，我们来讨论一个返回值非空、并且有若干个参数的函数。程序2-10给出了第一个C++程序的又一个版本。它有一个带有两个double类型参数的函数computeSquare()，函数的返回值类型为double。

程序2-10　有三个服务器函数的第一个C++程序

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

void displayInitialGreeting()                          // void return type
{ cout << "Welcome tc the C++ world!" << endl; }

double computeSquare(double x, double y)               // non-void return
{ double z;                                            // a local variable
  z = y + 1;
  y = pow(x,z);
  return y; }                                          // mandatory return statement

void displayResults(double y)                          // function header
{ cout << "In that world, pi square is " << y << endl;
   cout << "Have a nice day!" << endl; }               // function body

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();                            // function call
  y = computeSquare(x,y);                              // another function call
  displayResults(y);                                   // yet another call
  return 0;
     }
```

请注意对参数的类型和返回值的类型是没有限制的。上例中，参数与返回值的类型都是double类型，这纯属巧合。

请注意每一个参数都通过它们的类型和名字来指定，就像变量的定义形式一样。即使参数是同一个类型，每个参数都必须进行单独的描述。不能用逗号来分隔同一个类型的参数的定义，例如以下函数定义的参数表就是错误的：

```
double computeSquare(double x, y)    // syntax error
{ double z;
  z = y + 1;
  y = pow(x,z);
  return y; }
```

注意　参数和返回值类型可以是任意的。每个参数的类型都必须单独指定，即使它们属于同一个类型。

由于函数computeSquare()的头部指定了一个非空的返回值（这里是double），因此，函数体中必须有一个return语句。该语句使用了关键字return，并且以一个double类型的值作为返回的参数。有一些程序员将返回值放在括号中，但这并不是必须的。函数computeSquare()的函数体中定义了一个局部变量z，并对它赋值，然后计算变量y的值，再将该值返回给调用函数。（在第5章中，将会详细地介绍这一代码的几个微妙之处。）当客户代码调用这一函数时，会将实际参数的值传递给它，然后，这些值就可以在被调用的函数内部使用了。由于被调用的函数有返回值，因此，该返回值就可以像该类型的一个值那样，用在客户代码的表达式中（在本例中，作为一个double类型的值）：

```
y = pow(x,z);
```

请注意区分形式参数与实际参数之间的不同。*形式参数*（parameter）是在函数头部定义并且在函数体中使用的变量，而*实际参数*（argument）是在客户函数中定义并且在函数调用中使用的变量。通常我们使用术语形参（formal parameter）和实参（actual argument）。

为了简化起见，函数computeSquare()的形式参数和实际参数使用了同样的名字。这在实际生活中并不常见。在大型的程序中，客户函数由一个程序员开发，而服务器函数又由另一个程序员开发。他们之间不需要讨论参数的命名，客户的程序员只需要知道参数的类型和返回值的类型。

通常，服务器函数会先于客户函数开发出来，并放在库中，不可以直接得到其源代码。即使可以得到其源代码，但是对于客户的程序员来说，研究服务器的源代码从而知道形式参数的名字会加重他们编程的负担。幸好并不需要那么做。实际参数的名字与形式参数的名字没有任何的关系。以下就是体现了这种无关的和可用于程序2-10的函数computeSquare()的另外一个实现版本：

```
double computeSquare(double base, double exponent)
{ double power = exponent + 1;              // a local variable
  return pow(base,power); }                 // return statement
```

在上述所有的例子中，将所有服务器函数的定义放在调用这些函数的客户函数的前面。这与一般变量的用法类似——变量要先定义后使用。如果在源文件中改变这些函数出现的次序，会怎么样呢？这里有一个C++从C那里继承下来的限制，即如果预先没有进行函数的定义就调用了函数，编译程序将无法识别标识符displayInitialGreeeting()，并显示出错信息，如程序2-11所示的程序的错误版本。

程序2-11 函数定义在函数调用之后的不正确的C++程序

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();                 // syntax error
  y = computeSquare(x,y);                   // another syntax error
  displayResults(y);                        // and another syntax error
  return 0;
    }
```

```
void displayInitialGreeting()                          // function definition
{ cout << "Welcome to the C++ world!" << endl; }      // body

double computeSquare(double base, double exponent)
{ double power = exponent + 1;                         // a local variable
  return pow(base,power); }                            // return statement

void displayResults(double y)
{ cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl; }                // function body
```

本书使用的编译程序会试图给出有关项和函数重定义的“帮助”信息：

```
Compiling...
c:\data\ch02.cpp
c:\data\ch02.cpp(7) : error C2065: 'displayInitialGreeting  : undeclared identifier
c:\data\ch02.cpp(7) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(8) : error C2065: 'computeSquare' : undeclared identifier
c:\data\ch02.cpp(8) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(9) : error C2065: 'displayResults' : undeclared identifier
c:\data\ch02.cpp(9) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(13) : error C2371: 'displayInitialGreeting' : redefinition;
   different basic types
c:\data\ch02.cpp(17) : error C2371: 'computeSquare' : redefinition;
   different basic types
c:\data\ch02.cpp(23) : error C2371: 'displayResults' : redefinition;
   different basic types
DEMO.EXE - 9 error(s), 0 warning(s)
```

而不同的编译程序可能会给出某些不同的信息，但这并不是问题所在。通常，刚刚开始编写C++程序的程序员都会觉得难以相信：“标识符未定义是什么含义呢？”明明已经定义了标识符，难道编译程序看不见吗？编译程序并不是没有看见，这只是编译程序的设计者与编译程序的用户的观点不同。编译程序的用户忘记了一条最普遍的规则：代码中所有的东西都必须先定义后使用。这里所有的东西中包括了变量的名字以及函数的名字。

这种对函数调用的要求，其目的显然是：编译程序要检查函数的名字是否拼写正确、函数调用中是否给出了正确的实际参数个数、以及实际参数的类型是否正确。当然，出错信息最好能够提供更大的帮助，最好能够不向程序员提出疑问就能纠正错误。若能像一些早期的语言那样对源代码进行第二遍扫描，将会减少上述问题的发生。

在C++中，要求程序员使用函数原型来解决以上的问题。函数原型的语法形式与函数头部一样，惟一的不同是函数原型必须以分号结束。对函数原型的惟一限制是它必须位于它的第一次调用之前。通常将原型与客户函数一起放在源文件的开头。

函数定义与函数原型之间的关系，是变量定义与变量声明之间的关系的推广。函数原型声明了一个函数，它可以根据需要重复多次，而函数定义在程序中只能出现一次。

增加了函数原型的使用以后，就不再需要像客户函数那样将函数定义都放在同一个文件中。原型本身就可以通过编译程序的检查。程序2-12是程序2-11的另外一个正确的实现版本，它将若干个服务器函数原型放在文件的开头，并将函数定义移到了另外的地方。

程序2-12　一个函数原型在函数调用之前的正确的C++程序

```
const double PI = 3.1415926;
```

```
void displayInitialGreeting();          // function prototypes
double computeSquare(double x, double y);
void displayResults(double y);

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();             // function calls
  y = computeSquare(x,y);
  displayResults(y);
  return 0;
}                                       // end of the function block
```

即使是这样一个简单的例子，我们已经可以从中体会到使用函数的好处。main()函数的代码不需要再考虑具体的工作细节是如何完成的，它只要表达必须干什么工作。如果想知道工作的具体实现细节，维护人员可以找到不同文件中的服务器函数的源代码。由于每个函数位于一个单独的文件中，因此，维护人员的注意力就不会被无关的细节所干扰。程序员甚至不必关心服务器函数所在的源文件的名字。然而，这些函数的目标代码必须与main()的目标代码连接起来。这里也省去了#include文件，它们只需包含在使用这些库函数的服务器函数的源代码中。客户端代码程序员不必了解其服务器函数使用了何种服务。

如果位于不同文件中的不同客户函数都使用了同一个服务器函数，那么每个文件都必须包含这个服务器函数的原型。通常，程序员将原型放在不同的头文件中，并把它们包含在实现客户函数的文件中。包含像iostream这样的标准库文件与包含程序员定义的头文件之间的区别是，后者要使用路径名以及双引号。程序2-13是将上述程序中的原型移到文件c:\data\cppbook\ch02\display.h后的另外一个版本。

程序2-13 将原型放在另外一个文件中的第一个C++程序

```
const double PI = 3.1415926;
#include "c:\data\cppbook\ch02\display.h"   // prototypes

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();                 // function calls
  y = computeSquare(x,y);
  displayResults(y);
  return 0;
}                                           // end of the function block
```

在函数原型中可以选择是否使用参数名字。编译程序并不会根据它们来检查函数调用的正确性，而只会用到这些参数的类型。参数名字可能对文档化有用。如果没有用到参数名字，文件c:\data\cppbook\ch02\display.h的内容就变成：

```
void displayInitialGreeting();
double computeSquare(double, double);   // no parameter names
void displayResults(double);
```

2.7 类

下面我们来讨论将函数和数据组合成为一个聚集数据类型的语法。C++对此的实现方法

是，提供关键字struct和class以及将各个组件合并成可以整体处理的聚集的规则。在类定义中，必须指定数据成员（数据域）的类型和名字，以及（访问这些数据成员的）成员函数的头部或成员函数体。客户代码可以将类的名字作为一个类型名来使用。这意味着我们可以定义属于这种类型的变量（即对象），将它们作为参数传递给函数等等。

考虑这样一个问题：将每天的时间表示为小时与分钟的组合，需要有一个能够存储时间数据并且显示所存储时间的对象，而客户代码需要两种时间的显示格式，例如军用时间格式（18:45）或者通常的时间格式（6:45 P.M.）。

这里，建立了一个具有两个整型数据域hours和minutes的类描述，可以在类以外访问这个类。它描述了类对象（实例和变量）的组成，这些类对象将会在稍后的程序中实现其定义和操作：

```
struct TimeOfDay        // keyword struct is used
 {
int hours;              // one data member: an integer
int minutes;            // another data member: also an integer
 } ;
```

对类名的习惯写法是让组成类名的每个单词的开头字母用大写。开花括号和闭花括号表示类的作用域，它们是将类中的内容与外界分隔开的边界。与其他使用花括号的情形不同的是，在闭花括号之后必须有分号。要知道C++既不是简明的，但也不是令人厌烦的。

类中数据域的定义与变量定义的形式是类似的。它们将类型与域名联系起来。当生成了一个类TimeOfDay的对象（实例或者变量）以后，就会根据每个域的类型分配相应的内存。

```
TimeOfDay time1, time2;      // two objects are allocated
```

当域的类型相同时，可以只用一个类型名来说明若干个域，且域名之间以逗号分隔。这与定义C++基本类型变量的形式是一样的：

```
struct TimeOfDay
 {
   int hours, minutes;       // two integer data members
 } ;
```

再一次提醒大家可以用这种语法来定义变量和类数据域，但不能用来定义函数的参数。

请注意数据域是私有的，不能在类以外访问它们。下一步，我们将给出为客户代码服务，访问类数据域的成员函数（或方法）。这些访问函数必须是公共的，以便客户代码可以通过使用它们来设置和显示两种格式的时间。例如，实现函数setTime()、displayMilitaryTime()以及displayTime()。函数setTime()必须有两个参数：一个是小时，一个是分钟，而其他两个函数没有参数，它们显示存放在对象中的数值。程序2-14给出了带有数据成员和成员函数的类的定义语法。

程序2-14　将数据和函数组合起来的第一个C++类

```
#include <iostream>
using namespace std;
class TimeOfDay {                              // keyword class is used
  private:                                     // keyword private makes data hidden
    int hours, minutes;
  public:
    void setTime(int hrs, int min)
```

```
    { hours = hrs;  minutes = min; }
  void displayMilitaryTime(void)
    { cout << hours << ":" << minutes; }
  void displayTime(void)
    { if (hours > 12)
        cout << hours-12<<":"<<minutes<<"P.M.";
      else
        cout << hours <<":"<<minutes<<"A.M."; }
} ;
```

函数setTime()将参数的值复制到对象的数据域（hours和minutes）中。函数displayMilitaryTime()显示hours和minutes。函数displayTime()检查hours（以军用时间格式）是否超出了12，若是，则从hours中减去12，并将所得的差与分钟以及P.M.标志一同显示出来；否则，则将hours与minutes以及A.M.标志一同显示出来。本例中使用了C++的iostream库。

这些成员函数在类之中，因而可以不受限制地处理类的数据：它们可以为客户代码设置或者访问数据的值。它们为了客户代码的需要而存在，并非为了类自身的需要而存在，因此这些成员函数被定义为公共的：它们可以被客户代码调用。为了调用这些函数，客户代码必须使用标准的C++定义变量的语法来定义类对象，在这些定义中，客户代码将类型的名字（如TimeOfDay）与变量的名字（如time1、time2）联系在一起。

这里将类的定义放在头文件c:\data\cppbook\ch02\time.h中。为了使用这个类，客户代码所在的源文件必须包含这个头文件。否则，编译程序将会认为TimeOfDay没有定义。程序2-15给出了一个客户代码的例子，它定义TimeOfDay对象，分析类函数的返回值，并把结果进行相应的格式化输出。（注意必须包含类的头文件。）

程序2-15 将数据和函数组合起来的第一个C++类的客户代码

```
#include <iostream>
using namespace std;
#include "c:\data\cppbook\ch02\time.h"

int main(void)
{
  TimeOfDay time1, time2;                  // class instances
  int hours = 19, minutes = 15;            // integer variables
  time1.setTime(7,35);
  time2.setTime(hours, minutes);           // initialize objects
  cout << "First time: ";
  time1.displayMilitaryTime();             // message to first object
  cout << endl << "First time: ";
  time1.displayTime();
  cout << endl << "Second time: ";
  time2.displayMilitaryTime();             // message to second object
  cout << endl << "Second time: ";
  time2.displayTime();
  return 0;
 }
```

这里，变量time1和time2的类型都是TimeOfDay。它们的定义与基本类型的变量的定义方式相同。从本质上来说，类定义扩展了C++变量的类型集合，并增加了一个新的TimeOfDay类型。尽管这种贡献不是太大，但类的便利却相当显著。这种类机制提供了巨大

的扩展语言的机会。客户代码与类代码之间的关系如图2-6所示。当客户代码要访问类的私有部分时（例如要设置时间或者显示时间），它并不直接访问这些数据（如图2-6中的虚线所示），而是调用成员函数setTime()、displayTime()以及displayMilitaryTime()（如图2-6中的实线所示），并利用这些函数来为客户代码访问类的私有数据。

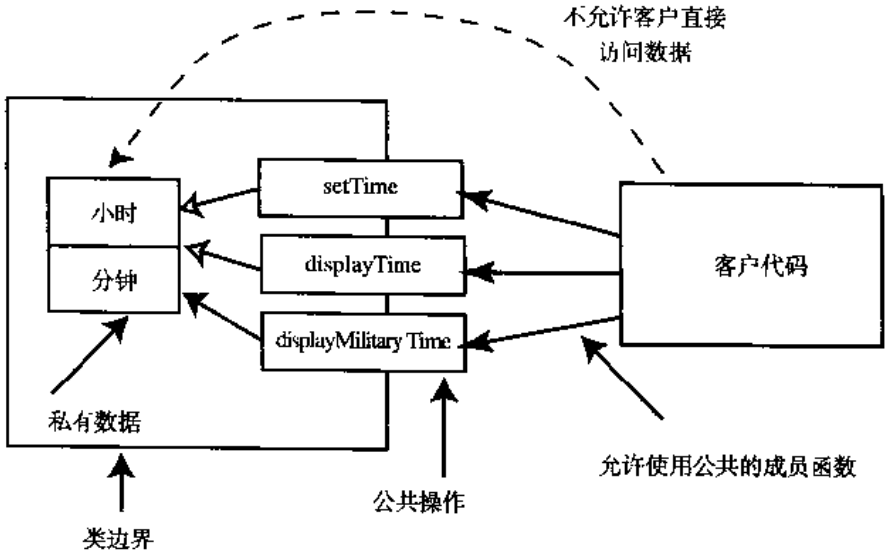


图2-6　使用访问函数而不是直接访问数据的客户程序代码

对成员函数的调用称为传递给对象一条消息。请注意消息的语法：在调用一个成员函数的时候，必须（使用点选择运算符“.”）指定函数要操作的对象。当目标对象是time1时，要显示的就是对象time1的hours和minutes的数据域；如果目标对象是time2，那么要显示的就是对象time2的hours和minutes域值。图2-7给出了程序的输出结果。

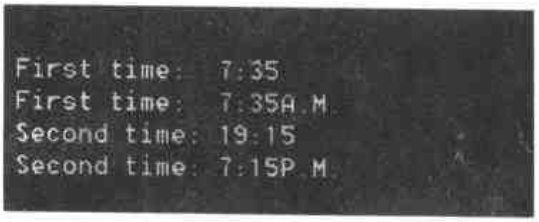


图2-7　TimeOfDay()访问函数的输出结果

与我们前面见到的其他函数不同，不可以只使用函数的名字来调用成员函数（另外如果有必要的话，给出所需的实际参数）：

```
displayTime();                    // syntax error
```

需要考虑的是：要显示什么时间？显示什么对象的时间？这里没有给出有关的说明，于是编译程序无法得知我们的使用意图。因此上述的函数调用是错误的。必须指出消息传递的目标，但是要注意该目标必须与被调用的成员函数的类型相一致，不能使用其他类型的变量作为目标。例如，以下的做法也是不正确的：

```
hours.displayTime();              // syntax error
```

变量hours是int类型的，可以对一个整型变量进行加、减以及其他的运算，但是不能对它进行displayTime()操作，因此该语句是不对的。目标必须具有正确的类型。

通过使用类复合以及类继承，就可以用类扩充C++语言的能力。在此，完全可以给出简单的类复合以及类继承的例子，但这样会使简介部分显得太长了。即使不给出这方面的例子，简介部分也已经达到了它的目的——已经让我们初步了解了C++语言，可以编写出简单的C++程序——尽管不太优美，但是已经足够强大，并且可以运行。更为重要的是，我们可以在这样一个坚实的基础上继续C++的学习。

在继续讨论语言的细节之前，我们应该了解与连接和执行C++程序有关的一些问题。

2.8 程序开发工具的使用

如前所述，我们可以使用一个文本编辑程序来建立一个C++源代码。用C++的编译程序对该代码进行编译，如果源代码中存在语法错误，编译程序就会给出有关的出错信息；如果没有语法错误，编译程序就让我们运行该程序。本节，我们将讨论一些更为复杂的生成和运行C++程序的工具。将会解释每一种工具的作用以及最有效地使用这些工具的方法。当然，这依赖于我们所使用的是与命令行相配的工具（如UNIX下的GNU编译程序），还是一个集成的开发环境（如Windows下的Microsoft IDE）。

许多开发环境要求（或者建议）除了要保存源文件以外，还必须将源文件插入到一个工程中（一个工程有若干个源文件）。作为一个整体，工程文件收集了有助于程序分析和调试的信息。另外，集成环境的供应商通常提供了可以生成多文件框架的功能，以便程序员可以将应用程序的代码加入到由这些工具生成的类和函数框架中。对于一个有经验的程序员来说，这是很有用的，但对于初学语言的人来说，这就是一个障碍。除了要关注语言的一些特定的问题以外，还必须研究由这些工具生成的代码，这些代码或许会用到一些初学者不太熟悉的语言特征。

提示 为了避免不必要的复杂性以及简化无关的工程管理任务，在使用语言做实验时，最好能够将程序都集中在一个文件中。一些集成环境不管程序员是否需要，都为其生成一个缺省的工程。程序员只需要接受它并逐渐熟悉这种环境，但在初学C++时应避免使用集成环境提供的代码，这将使学习简单化。

在编辑文件后调试程序前保存文件是一个好主意。否则，如果程序包含运行时错误并导致了系统瘫痪，那么将会丢失所做的修改。现在只有少数的人在输入代码修改之前先将其写在纸上，所以人们往往在系统瘫痪时就会丢失大量已完成的工作。在编译程序之前先将文件存盘就会减少这种危险。为了帮助程序员了解源文件的状态，一些编辑程序会在屏幕上显示有关的状态信息。如果修改了代码却没有存盘，编辑程序就会在编辑窗口的顶部的文件名之后加上星号或者在编辑窗口的底部给出提示信息。当文件存盘以后，星号（或者提示信息）才会消失。

而另外的一些集成环境不信任程序员具备的常识，在编译程序之前会要求保存源文件，即使程序员只是进行实验性的修改，并不确定保存所做的修改还是恢复源文件。

如果源文件没有语法错误，编译程序就会生成一个与源文件同名的目标文件。根据系统的规定，目标文件的扩展名可以是.o或者.obj。如果源代码含有语法错误，编译程序不会生成目标文件，而是给出出错信息，提示在程序的什么地方出现了什么类型的错误。使用早期的工具时，必须对源文件的行数或者依赖于编辑程序显示的行号。较新的工具已经不需要跟踪程序的行号，即使它们也会指出出错的行号。当程序员点击信息窗口中的出错信息时，工

具就会将光标直接定位在编辑窗口中出错信息所指示的行上，这是一个十分有用的特征。

当我们熟悉了语言以后，就会发现编译程序给出的出错信息是可以理解的，并有帮助，至少其中一部分是这样。在还没有熟悉语言的时候，请记住下面这条规则：千万不要对编译程序给出的出错信息咬文嚼字，至少在第一次遇到困难的时候要马上停止。不要拘泥于出错信息，不要坚持，否则将会徒劳无益。

至少有90%的出错信息对初学者来说是不可理解的。是否有例外呢？例外就是大约有10%的出错信息是容易被人理解的，但对于这种错误，我们只需要扫视一下其所在的代码行就会很快发现错误，也就是说，根本不需要出错信息的帮助。这就是建议不要将时间、精力和感情倾注到出错信息的分析之上的原因。在熟悉了有关的术语以后，我们就会从这些信息中获益，这时才值得花时间对出错信息进行研究并决定下一步的方向。但在开始学习语言的时候，研究出错信息却是弊大于利。

出现这种情形的原因是，编译程序的设计者希望能提供最大的帮助，于是就尽可能详细地分析所处的情形。因此，使用高级C++术语表达的出错信息对于初学者而言就很难理解了。通常，问题可能由几种不同类型的错误引起，但是编译程序既没有时间也没有知识对它们进行细致的分析，并精确地指出问题出现的原因。因此，看起来技术性很强、相当详细的出错信息，却与实际发生的错误没有太大联系。

可以坦白地说：不要期望从编译程序的出错信息中得到多大的帮助。最好还是依靠自己的智慧。惟一可以依赖的出错信息就是发生错误的位置，即使该信息也经常是不太可靠的。有时出错位置是在所指位置的前一行。因此，当得到出错信息时，不要只检查出错信息所指的代码行，也要检查它的前一行。

通常，编译程序会给出不止一个出错信息。当不清楚第一个出错信息的原因时，许多程序员会去分析第二个、第三个以及其他的出错信息，也许后者看起来更容易理解。

但千万不要这样做。如果不清楚第一个错误的话，就不要转去分析第二个错误。改正了第一个错误以后，也不要马上修改第二个错误。这是不对的。过去，编译一趟要花费几分钟甚至几小时的时间，由于时间这么长，把时间花在分析编译程序给出的出错信息上是有意义的（即使有一部分是虚假的出错信息），并且希望尽可能一次排除多个错误。

现在，编译程序只需几秒钟时间就可以完成编译工作。因此花时间去分析第二个、第三个错误是浪费的，因为当第一个错误改正了以后，其他错误可能已经发生了变化或者已经消失了。它们的出现可能并不是因为有错，而是因为第一个错误的出现而使编译程序无法跟踪源代码流程。程序员的时间比编译程序的时间更宝贵，追踪幻想的错误是没有回报的。随着经验的积累，程序员会更容易发现真正的错误，也就更容易一次改正若干个错误。但在达到这一境界之前，请不要太相信编译程序，只需一次分析一个错误，然后重新编译。

以下是一个含有某个错误的程序2-1的另外一个版本。编译程序给出了三条出错信息，分别指出了出错的行号以及出错的原因。

```
#include <cmath>                                    // preprocessor directive
#include <iostream>                                 // preprocessor directives
using namespace std;                                // compiler directive
const double PI = 3.1415926;                        // definition of a constant
int main(void)                                      // function returns integer
{
  double x=PI, y=1, z,                              // definitions of variables
  cout << "Welcome to the C++ world!" << endl;      // function call
```

```
    z = y + 1;                                          // assignment statement
    y = pow(x,z);                                       // function call
    cout << "In that world, pi square is " << y << endl;
    cout << "Have a nice day!" << endl;
    return 0;                                           // return statement
}                                                       // end of the function block
```

```
Compiling...
ch02.cpp
C:\Data\ch02.cpp(7) : error C2143: syntax error : missing ';' before '<<'
C:\Data\ch02.cpp(7) : error C2143: syntax error : missing ';' before '<<'
C:\Data\ch02.cpp(10) : error C2296: '<<' : illegal, left operand has type 'double'
C:\Data\ch02.cpp(10) : error C2297: '<<' : illegal, right operand has type 'char [29]
C:\Data\ch02.cpp(11) : error C2296: '<<' : illegal, left operand has type 'double'
C:\Data\ch02.cpp(11) : error C2297: '<<' : illegal, right operand has type 'char [17]
Error executing cl.exe.

ch02.exe - 6 error(s), 0 warning(s)
```

这里的全部智能检查活动都是徒劳的，就像为了节省半底鞋而在泥浆中爬行去寻找丢失在另一隅的胶鞋一样。我们可能会跟踪执行的轨迹，并试图在第7行的<<运算符之前加上分号，或者想知道第10、11行的操作数的类型，所有这些都出现在有错误的地方。因为不管编译程序如何提示，第7、10、11行上都没有语法错误。错误只是在第5行，它将分号错写为逗号。

在此，还想再讨论一下关于编译程序的出错信息，程序员会经常犯这类小而简单的错误，尽管我们并不期望它们的出现。它们与误导的出错信息在一起就更难发现。通常，会怀疑其他语句，特别是当我们对那些语句的语法有某种怀疑的时候。于是我们就会修改那些语句，这可能会进一步造成更多的错误并出现更多的出错信息。然后，我们会继续分析新的出错信息、继续修改，使得程序的调试情况越来越糟。当花费了大量的时间最终将问题解决了以后，我们不禁会感叹："我们为何会犯这么愚蠢的错误？为何在如此明显的情况下本来可以检查源代码而没有检查它？我怎么会这么笨？"

因此，我要补充对编译程序出错信息不要咬文嚼字的建议。不要基于这些出错信息来评价程序。毕竟，最终一定会发现错误并修改它们。花费时间去分析这些出错信息虽然重要，但并非十分重要。对自己的编程技巧有信心是更重要的，而不断的自我批评只会削弱自信心。另外，这种责备是完全不公平的。不要责备自己。要有自信并相信自己的能力。

有时编译程序会给出警告信息。当我们有了足够的经验以后，最好采用与分析错误信息的策略相同的策略去分析它们，因为这些警告信息通常反映出程序存在的错误。但是在没有足够的经验以前，忽略警告信息，它们通常有误导成分。由于警告信息不会妨碍编译程序生成目标代码、运行和调试代码，因此可以在运行时测试代码，但随后必须理解并消除警告信息。

成功编译后的下一个步骤就是连接，有的工具也叫"建立"（building）。当程序由若干个源文件组成时，在开发阶段每个文件都可以独立地进行编译。如果源代码中含有在其他文件中定义的标识符声明（变量或者函数），编译程序就会不知道这些标识符的地址，因为编译程序一次只编译一个文件。当所有源文件都编译成功时，就可以将它们连接起来。连接程序就会检查所有的目标文件，并解决在其他文件中定义的标识符的外部引用。

在连接阶段，连接程序将更多C++库中一些已经预编译的代码加入到目标代码中。尽管库源代码通常都是可用的，但对函数调用pow()、运算符、<<或其他库函数都不会重新编译。已经预先编译好了库函数。连接程序通常以同样的方式来解决这些外部引用问题。

连接的结果是可执行的程序文件。如果是一个单文件程序，可执行程序的名字就与源文件的名字相同。如果是一个多文件程序，可执行程序的名字就与该工程的名字相同。（程序员还可以选择任意的名字。）通常，可执行程序文件的扩展名是.exe。

连接错误并不经常出现。当它们出现时，出错的原因或者是误写了函数的名字，或者是对工程的不正确的操纵。

可执行程序可以运行。大多数IDE可以让程序员选择是在调试模式下还是在产品模式下运行程序。最好使用调试模式，但建议不要使用调试程序。在开始学习C++的时候，不可避免地要忙于学习语言、编辑程序、编译程序以及连接程序的使用。而学习调试程序是主要的困难，在开始编写复杂的C++程序之前，花费这么多的时间学习却得到很少的成效。因此，目前只需要检查程序的输出就足够了（如果需要，可以增加一些打印语句）。

另外，还值得提醒的是：程序员通常在见到程序的输出时，无法判断结果是否出错。对此，本书无法作出很好的解释，但这是事实。这可能与人的自欺和自制方面的天性有关，无法确定其真正的原因。不管是什么原因，通常我们会忽略这些运行程序时的错误。

一些程序员为了避免上述情形的发生，会预先将程序运行的期望结果记录下来。这会有一定的帮助，但却并不完全保险。

请在检查程序的输出结果的时候保持足够的警惕。

2.9 小结

祝贺大家已经学习了C++程序设计语言中的最重要的部分！我们已经了解了可以使用什么预处理程序指令，如何定义变量和函数，如何通过条件语句和循环语句控制执行流程，如何对代码进行注释，如何忽略误导的编译程序出错信息等方面的知识。甚至已经看到了如何定义和使用一个C++的类。这实在是太好了！

实际上，这对于编写大多数所需要的C++代码来说已经足够了。与学习自然语言类似，不纯的洋泾浜英语比英语要小型并且简单，但吸引人的是，对于一个有经验的使用母语的人来说，只使用语言的一小部分就能表达众多的内容。这里可能再一次印证了一条规则：80%的工作是由20%的人来完成的。

但是，我们不能只局限于学习语言20%的部分，开始学习语言的其他内容，以便成为有经验的程序员的时候到了。

第3章 C++数据和表达式的使用

本章里，我们将学习如何使用C++数据：什么样的数据类型是可用的，这些数据类型上的值支持什么操作，以及C++ 程序员应该知道有什么易犯的错误。就像这种语言的其他许多方面一样，C++ 优缺点共存。它的数值数据类型集合很小，而且现有类型之间的差别也不是那么大。所以，在它们之间进行选择总不是那么清楚。它的运算符集合很大。一些 C++ 运算符相当复杂，还有另外一些则有着不常见的记号。C++ 数据类型和运算符都存在可移植性方面的潜在问题，程序在不同机器上的运作情况通常是不同的。

C++ 继承了 C 语言巨大的灵活性：它可以把值从一种类型转换为另一种类型，并把它们组合为复杂的表达式。下面就来讨论什么是可用的。

3.1 值及其类型

在C++语言中，每一个值在其生命周期（程序执行期间）的每一刻都以其类型作为特征。C++变量在定义时便指定了其类型。类型描述了数值的以下三个特征：

- 该类型的值在计算机内存中的大小。
- 对于该类型来说，合法的取值范围（表示了该类型值的位模式的解释方法）。
- 该类型值上合法的操作集。

举个例子，int类型的值在作者的计算机上被分配了4个字节，则其合法值的取值范围是从–2 147 483 648到+2 147 483 647；其合法的操作包括赋值、比较、移位、算术运算以及其他一些运算。在第2章2.7节“类”中，所定义的TimeOfDay类型的值被分配了相当于两个整数大小的内存空间（除非为了更快地存取，编译程序要在内存增加空间来排列这些值）。TimeOfDay集合上的合法取值就是第一个整数（从0到23中取）与第二个整数（从0到59中取）的任意值的组合。TimeOfDay集合上的合法操作包括了setTime()、displayTime()和displayMilitaryTime()；它还包括赋值运算，但不包括比较运算。当然，是TimeOfDay的成分而不是TimeOfDay的值可以进行比较运算（它们是整数，可以对它们使用int的规则）：我们应该能够区分类型的属性和其成分的属性。如果客户代码一定要比较TimeOfDay的值，类TimeOfDay就要通过实现诸如isLater()或compareTime()或类似的函数才能支持。（请注意这里的客户/服务器术语）

每一个C++变量的定义都必须指定其值的类型。另外，类型也可以用来说明常量、函数和表达式的取值特征。这就意味着可以把某类型上的值组合为一个表达式，从而得到其他类型的值，并把这些值用于其他的表达式中。

在大多数情况下，类型是用标识符来表示的。也就是说，类型有一个名字（比如int或TimeOfDay）。这是通常的情况，但这并不是定义类型的惟一途径。C++ 允许所谓的匿名类型，这些类型没有指定名字，虽然这并不常用。

C++ 的基本类型名有以下的保留字：int、char、bool、float、double和void（实际上，这是保留字的全部清单）。在这个清单中，void表示可以在一个表达式中操纵的空

值，用它指出在其他表达式中使用的数值是不适合的。比如说，在第2章的2.6节“函数和函数调用”里的函数computeSqaure()，它返回的值可以用在表达式中；在同一节里的另一个函数displayResults()却不能这样使用：它没有返回值。如果试图不正确地使用它的值，编译程序将会指出这是一个错误。

```
int a, b;
a = computeSquare(x,y) * 5;          // this is legal C++
b = displayResults(PI*PI) * 5;       // this is an error
```

其他语言没有这种特殊的“类型”，因为它们有函数（有返回值）和过程（没有返回值）之分。C++ 继承了C的函数语法：它既可以是函数也可以是过程。从逻辑上来说，没有具体的返回类型应解释为没有返回值；但在C中不是这样。为了避免伤害，在C中没有指定类型就是指整数类型，并要求有一个返回整型数值的return语句。C++对此使用了折中的方法。如果没有指定返回类型，编译程序不会继续下去，也不会（像C编译程序那样）要求函数返回一个整型数值。新的C++ 编译程序假定程序员想要的返回类型是void。

```
displayResults(double y)                          // C++ it is void
{
  cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl;             // no error in C++
  }
```

然而，如果把这个函数当做一个操作数用在表达式中，C++ 会假定使用的是老的C语言的规定，即程序员希望返回的是一个整数。运行时，diplayResults()会无警告地返回无用数据。正如语言设计者所言，编译程序“不会再猜测程序员的意图”并去掉了编译时的保护。

```
b = displayResults(PI*PI) * 5;       // not a syntax error
```

如果给出了return语句，没有返回类型的函数会被看做返回了一个整型数值。

```
displayResults(double y)             // C++ assumes it is int
{
  cout << "In that world, pi square is " << y << endl;
  cout << "Have a nice day!" << endl;
  return 0;                          // no syntax error
  }
```

如果觉得适合，客户代码可以使用返回值。

```
b = displayResults(PI*PI) * 5;       // this is legitimate
```

将int用作缺省的返回类型，是因为大多数的C函数都被设计为有返回值，能让程序员输入程序时节省3个按键被认为是一个很重要的优点。要避免这一做法。如果返回类型是整数类型，就应写为int；如果一个函数没有返回值，则应将返回类型指定为空值。

警告 通常要指定函数的返回类型。如果该函数没有返回值，则把类型指定为void。不要依赖C++ 的缺省类型。

除了C++的基本类型以外，程序中定义的类型被称为用户自定义类型。但实际上，因为用户并没有定义类型，用户是通过系统的实现来达到预定目标的个人或组织。定义类型结构和类型名字的人是程序员，比如第2章2.7节“类”中的TimeOfDay类型。因此本书把这些类型称为程序员定义类型。

虽然C++中不同的类型所占的内存空间大小是不同的，但不同类型的值占用相同大小的内

存空间也不奇怪。对于不同的类型来说，是通过对位模式的解释来区分它们的值。例如，如果存储在一个整数变量中，那么位模式01000001就被解释为65；如果存储在字符类型的变量中，则同一个位模式就被解释为A。

过去，程序员必须学会理解二进制数、八进制数、十六进制数、ASCII编码和EBCDIC编码，必须把2^{16}熟记于心（有时是2^{20}，或甚至是2^{32}），还要理解关于负数的补码和反码表示法以及其他一些难以理解的东西。今天，大多数程序员不再需要这样做了。在容量方面，计算机硬件仍然以8位的倍数建立。一个字节占8位，半个字占16位，一个字占32位。在某些计算机上，一个字占16位，一个双字占32位。因此至少能够知道可以存储在不同大小的内存中的数值的范围。

因此，4位（十六进制的1位）可以表示16个不同的位模式。通常，这16个位模式可以用来指定从0到15的整数。类似地，8位可以表示256个数值（2^8）。这256个位模式可以用来指定0~255的整数。如果我们想同时表示正数和负数而不仅仅是正数，情形会如何呢？注意我们仍然只有这256个位模式。要表示-128~+128这个范围是不行的，因为这里有257个值，而不是256个。常见的表示范围是-128~+127。

两个字节（16位）可以表示65 536个位模式（这个数是2^{16}）。对正数来说，其范围是0~65 535。对于有符号的数值（正数和负数）来说，其范围是-32 768（2^{15}）~+32 767（$2^{15}-1$）。类似地，32位（4个字节）可以表示4 294 967 296个数值。对于有符号的数来说，4个字节可表达的数值范围是-2 147 483 648（2^{31}）~+2 147 483 647。这是我们应该知道的有关二进制数的知识。

3.2 整数类型

在所有的计算机体系结构中，C++的整数类型是最基本的类型。“基本”是什么意思呢？它意味着在给定的平台上，这一类型的值总是能最快地执行其上的操作。表示这一类型的关键字是int。

```
int cnt;
```

int的大小决定了可以表达的数值范围（2的位数次幂）。现在，业界已从16位结构转向32位结构，但在今后的一段时间里，这两种结构都还会用到。大多数固定的设备都会用32位计算机，但嵌入式系统和通信系统还将继续使用16位计算机，并且随着计算机应用不断普及到汽车、主要电器装备、甚至是烤面包机等领域后，这些使用16位机系统的数目还将上升。

这意味着为某一结构编写的程序在另一结构上运行时可能会不完全相同。

如果在可存储一个整数的数值中无法容纳某个整数的话，会发生什么事呢？其答案是：不会有大问题。在C++ 里没有诸如算术溢出的事件。能在一台16位计算机上从1加到32 767吗？尽管去做吧。结果将会是-32 768。如果想再加上1，其结果会是-32 767。

程序3-1给出的是一个在16位平台上运行的程序（该平台由一个32位机以及一个16位编译程序组成）。头文件limit包含了在给定平台上与实现有关的一些数值常量，常量INT_MAX就是这样的数值（32 767）。在这个例子中，使用了while循环以及iostream库（while循环类似于在第2章中讨论过的那个循环）。这个程序的输出如图3-1所示。变量num可以随着时钟面变化，并假设为负值。在cout语句中的每一个元素都有自己的输出运算符<<；甚至包括了在打印变量cnt和num之间的打印间隔（在双引号内）。

程序3-1　整数溢出的例子

```
#include <limits>
#include <iostream>
using namespace std;

int main(void)
{
  int num = INT_MAX - 2;
  int cnt = 0;
  cout << "Integer overflow in C++:" endl;
  cout << "Incrementing from " << num << endl;
  while (cnt < 5)
  { num = num + 1;
    cnt = cnt + 1;
    cout << cnt << "    " << num << endl; }
  cout << "Thank you for worrying about integer limits" << endl;
  return 0;
    }
```

```
Integer overflow in C++:
Incrementing from 32765
1    32766
2    32767
3    -32768
4    -32767
5    -32766
Thank you for worrying about integer limits
```

图3-1　整数溢出没有终止程序；它默默地产生不正确的结果

早期的C++（和C）版本不允许使用运行时的值来对变量进行初始化，变量的值必须在编译时确定。然而，不仅可以使用一个具体的数值，还可以使用一个表达式（例如程序3-1中的INT_MAX-2）来初始化变量。在现在的C++版本中，初始化表达式可以是任意复杂性的，甚至可以包括运行时函数调用的返回值。例如下面的例子在C++中是合法的。

```
int a = computeSquare(x,y) * 5;        // this is legal C++
```

从编译程序设计的观点来看，这是一个重大改进。早期的C和C++ 编译程序不支持这一特征。现在大家是否会觉得这过于重复呢？在3.1节“值及其类型”中，不是说过在计算中可以使用函数的返回值吗？

```
a = computeSquare(x,y) * 5;     // legal in C and C++
```

请注意它们的区别。在3.1节“值及其类型”中的例子说明的是赋值的情况。在C++ 、C或其他任何语言中，它总是可行的。本节中的这个例子说明的是初始化。尽管代码非常相似，但它们是两件不同的事情。初始化会分配内存并设置其值；而赋值所处理的对象(变量)已分配了内存，在内存具有其地址（并可能在该地址上有初始值），赋值只是改变了该地址上的值。在第2章曾提过这个区别，不久将会看到它更深的含义。

尽管编译程序的设计有了这一改进，C++并不期望其编译程序是一个两遍扫描的编译程序，它们都是一个一遍扫描的编译程序，没有能力预见未来。这就是为什么它们不能使用未定义值的原因，即使这个值在下一行定义。例如，下面的语句是错误的。

```
int a = b, b(5);                    // error in C++
```

在这里，变量b不能用来初始化变量a，但相反的次序则是合法的。（注意，初始化的语法类似于函数调用的语法，这在C 中是不允许的,但在C++ 中却是允许的。）

```
int b(5), a = b;                    // this is acceptable
```

3.2.1 整数类型修饰符

C++继承了C 的一种能很好的调整整数范围的技术：使用修饰符（qualifier）。以下是一些能够调整整数所占用存储空间的大小、或者能够改变对位模式解释的关键字：signed、unsigned、short和long。

我们一直都将修饰符signed作为 一种缺省情况来使用，因此它不必明确地指定。变量cnt的下面的定义和以前的定义完全等价：

```
signed int cnt;                     // signed is default
```

修饰符unsigned可以用来修饰那些不能取负值的变量（如下标、计数器、标签、目录数量等）。这个修饰符不会改变为其值所分配的内存大小（16或32位），但会改变对其位模式的解释。在16位机上，unsigned整数的合法取值范围不是-32 768~+32 767，而是0~65 535；在32位机上取值范围则是0~4 294 967 295。程序3-2给出了对前一例子用unsigned整数代替signed整数后的另一个版本。这一版本的输出如图3-2所示。可以看到前面出现的那个问题消失了。当然，当处于unsigned整数的上界时，它又会出现。只是出现的方式不同而已。当unsigned整数溢出时，将不作任何提示地回到0而不是一个大的负数。因此，不能相信这个版本会更好。

程序3-2 unsigned int类型的例子

```
#include <limits>
#include <iostream>
using namespace std;

int main(void)
{
  int unsigned num = INT_MAX - 2;
  int cnt = 0;
  cout << "Integer overflow in C++:" << endl;
  cout << "Incrementing from " << num << endl;
  while (cnt < 5)
  { num = num + 1;
    cnt = cnt + 1;
    cout << cnt << "    " << num << endl; }
  cout << "Thank you for worrying about integer limits" << endl;
  return 0;
    }
```

使用unsigned整数是一个好主意（虽然并没有将值的范围扩大很多，但这样做可以把设计者的意图传达给维护人员），它表示了指定的变量不能取负值。另一方面，如果维护人员没有注意到这一意图而使unsigned整数变量取了负值，就会引起灾难性的结果。程序3-3是前一个程序的另一版本，它令变量num的初始值为2并且在循环中不断地减1。程序的输出如

图3-3所示。

```
Integer overflow in C++:
Incrementing from 32765
1    32766
2    32767
3    32768
4    32769
5    32770
Thank you for worrying about integer limits
```

图3-2　当内存大小相同时，对于unsigned整型数值，溢出值发生在比一般整数的溢出值还大的数值上

程序3-3　对一个unsigned变量赋给负值

```
#include <iostream>
using namespace std;
int main(void)
{
  int unsigned num = 2;
  int cnt = 0;
  cout << "Negative values in an unsigned variable" << endl;
  cout << "Count down starting with +1" << endl;
  while (cnt < 5)
  { num = num - 1;
    cnt = cnt + 1;
    cout << cnt << "    " << num << endl; }
  cout << "Thank you for worrying about integer limits" << endl;
return 0;
    }
```

```
Negative values in an unsigned variable
Count down starting with +1
1    1
2    0
3    65535
4    65534
5    65533
Thank you for worrying about integer limits
```

图3-3　unsigned变量不能具有负值；进行减1运算时，在没有警告的情况下出现一个很大的正数

有两个修饰符可以控制一个整数所分配的内存大小，它们是long和short。

```
int cnt;  short int short_cnt;  long int long_cnt;
```

这里的目的不仅是为了给整数提供更大的取值范围，而且是为了节省所保存的空间。C++程序员通常关心程序的性能，即执行的时间和空间。使用signed整数（无修饰符时）可以提供最快的数据类型，使用long整数可以避免溢出但也占用了更多的内存，而使用short整数可使程序员避免浪费内存。比如，前面例子中的变量cnt，它的取值范围是0~5，在现代计算机上为何分配给它32位呢？实际上，一个字节就已经大大超过要求了。对于内存紧缺的计算

机来说，这一变量用short整数类型会更好。

用short修饰符来节省内存和用long修饰符来扩展数值的取值范围有多重要呢？这些修饰符使程序更加复杂。只有在有溢出问题（或内存紧缺）存在并且使用修饰符可以解决该问题时（通常两个情形都不是），程序员才会使用它们。否则，大多数程序员都使用不带修饰符的常规的整数，并且不必担心这些问题。特别是在现代的32位计算机上。用4个字节表示常规的整数已经可以避免以前的溢出问题。拥有充足的内存就没有必要再用short整数来节省空间。

由于通常的C++特征都是从C继承而来的，因此情况可能并非我们所想像的那样。从逻辑上说，short整数所占用的内存应该比整数少，而long整数所占用的内存应该比整数多。然而C（和C++）的标准只要求编译程序的设计者做到：short int所占用的内存不比常规的int多，以及long int所占用的内存不比常规的int少。这听起来并不会令人混淆。在16位的计算机上，short int和int的变量都会分配相同大小的存储空间，即16位，而long int的变量会分配32位。在32位的计算机上的情况刚好相反：short int数值分配了16位，而int和long int则分配了32位。

C++中有一个以字节为单位来计算数据所占的存储空间的sizeof运算符，其参数可以是一个变量名或者是一个类型名。对于任何平台，sizeof运算符的返回值之间都满足以下关系：

```
sizeof(short int) <= sizeof(int) <= sizeof(long int)
```

这一设计有一个有趣的结论：不管计算机是16位还是32位，short int和long int所占用的内存大小总是不变的。在任何机器上，short int总是16位，而long int总是32位。这就是为什么那些关心可移植性问题的程序员通常不使用常规的整数类型的原因。对于相对小的数值他们会使用short int，而其他不适合用作short int的数值就都说明为long int。他们通常是一些设计嵌入式系统和通信系统的程序员。在这些系统中，由于大小和价格的限制，内存通常都会超过市价，同样的代码必须能够在多种硬件平台上运行。

提示 整数是16位还是32位是由硬件决定的；但短整型数总是16位，长整型数总是32位。使用它们就可以解决可移植性问题。

是否可以像下面的例子那样将修饰符unsigned和修饰符short以及long合在一起使用呢？

```
unsigned short int short_cnt;  long unsigned int long_cnt;
```

是的，这是可行的。（注意，修饰符的顺序没有关系。）例如，在硬盘控制程序中可以找到这种用法：在那里，文件的大小或柱面的数目要求为非负大整数。然而，对大多数应用来说，使用常规的整数是避免产生额外复杂性的好方法。

另外，在本章的开头，曾提过这样一个旧规则：如果省略类型名的话，缺省的类型就是整数类型。这一规则仍适用于目前的情况。当使用long和short数据类型时，不必写出关键字int。

```
int cnt;  short short_cnt;  long long_cnt;        // same meaning
```

整型数字面值可以使用十进制、八进制及十六进制的形式来表示。比如，十进制数的64可表示为八进制数的100或十六进制数的40。为了避免混淆，书写形式上以0开头的数就表示八进制数，以0x（或0X）开头的数就表示十六进制数。所以，100意味着100（十进制数）；

但0100是八进制数，它表示十进制数64；而0x100是十六进制数，它表示十进制数256。

字面值在内存中的分配方式和变量的情形非常类似。惟一不同的就是我们不能操纵它们的地址，因此不能改变所存储的值。所以，63可以作为一个short类型的数值分配两个字节，也可作为一个long类型的数值分配4个字节。为了区别这两种情形，我们可以使用以下形式的大写或小写的修饰符来表示short和long常量：63s、63S、63l、63L。对unsigned类型的数值也可以这样表示：63u、63U、63us或63UL。这在实际中不太重要。

3.2.2 字符

C++ 把字符类型当做是另一种整数类型。它的大小为1个字节（8位），它可以表示任意的ASCII符号：字母、数字或非打印控制字符。下面是定义字符类型变量的例子：

```
char c, ch;  char first, last;
```

字符没有修饰符short和long。然而，修饰符unsigned和signed允许用于字符。不幸的是，其缺省类型还未形成统一的标准。在某些计算机上，char类型表示unsigned char；而在另一些计算机上，char类型则表示signed char。

通常来说不必过多考虑这些问题。因此也就没有形成统一的标准。然而，当把char值用作整数来进行计算时，上述的差别就会显得重要了。比如，一个signed char可以包含表示文件结束的库常量EOF，其值定义为-1；而unsigned char只能包含正数。所以，如果试图把-1放到unsigned char类型的变量中，就会发现所存放的内容是255，而不是-1。由于char类型既可以是unsigned也可以是signed，因此会引起可移植性问题。

与其他变量一样，char变量可以在定义时进行初始化，或者将来再赋值。可以用小整数来进行初始化和赋值，而它们的值会被解释为字符的编码。字符在字面上必须写在单引号内，它们可以是字符、八进制或十六进制数值或转义序列等。注意不要混淆单引号和双引号。单引号用来表示字符的内容，而双引号用来表示字符串（字符序列或数组）的内容。

```
char c = 'A',  ch = 65;          // both c and ch contain 'A'
```

这个例子用写在单引号内的字符以及十进制数形式的字符表示字符。其他表示字符方法是以转义字符'\'开头。转义字符不是一个普通字符，其作用是告诉编译程序要用特殊方式处理随后的内容。比如，把随后的数看做是八进制或十六进制数值。

```
c = '\0101';  ch = '\0x41';     // octal and hex values for 'A'
```

此处并不是真正需要引号和转义字符；八进制数可以直接表示为以0开头的数值、十六进制数可以直接表示为以0x（或0X）开头的数值。

```
c = 0101;  ch = 0x41;           // octal and hex values for 'A'
```

只有在需要将值嵌于串中的时候，转义字符才是必须的。另外，使用转义字符也会向维护人员表明：正在处理的是字符而不是数字。最常见的转义序列是换行符'\n'。其他的转义序列有'\r'（回车）、'\f'（换页）、'\t'（制表符）、'\v'（垂直制表符）和'\a'（响铃）等。

由于单引号和双引号在C++中有着特殊的作用，于是必须使用转义字符来表示它们：即'\''和'\"'。对转义字符自身来说也是一样，如果需要显示它，就要在一行中使用两个转义字符：'\\'。

还有两个转义序列：'\b' 和 '\0'。第一个表示后退一格。第二个表示数字0；它不是一个可打印的字符（可打印的0的数值为48），但却是一个很重要的字符。在每个串的结尾，C++编译程序都会插入这个字符，它用来标识串的结尾。比如，字符串 "Hello" 中包含了6个字符而不是5个；其最后的一个字符是编译程序所插入的 '\0'。由于还未讨论数组，在此还不能更详细地讨论字符串，但不久将会更详细地讨论。

C++把字符当做小整数。因此，可以在字符值上进行算术运算和比较运算。程序3-4给出了这种字符操纵的例子。程序首先用大写形式打印出字母表，然后用小写形式打印出字母表。它也给出了用转义字符来输出单引号、双引号和转义字符自身的方法。程序的输出如图3-4所示。

程序3-4 操纵字符数值的一个例子

```
#include <iostream>
using namespace std;
int main(void)
{ char ch;  int cnt;
  ch = 65; cnt = 0;                  // ch contains 'A'
  while (cnt < 26)
    { cout << ch;
      ch = ch + 1; cnt = cnt + 1; }
  cout << endl;
  ch = 'a' - 1;                      // ch contains character '`'
  while (ch < 'z')
    { ch = ch + 1;                   // ch contains 'a', 'b', ... 'z'
      cout << ch; }
  cout << endl;
  cout << "Single \' and double \" quotes are special\n";
                                     // new line: same as endl
  cout << "And so is the escape character \\" << endl;
  return 0;
  }
```

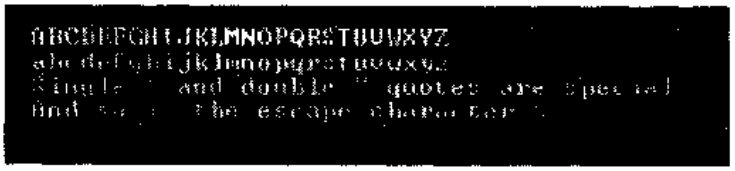


图3-4 对字符变量进行算术运算的程序（程序3-4）的输出结果

对字符变量进行算术运算不是一个好做法，因为它会使维护人员感到困惑。另一个问题是它只能在具有连续字符集（如ASCII码）的计算机上工作。如果用在不具有连续字符集（如EBCDIC码）的计算机上，程序3-4的代码将产生一些不可打印的结果。然而，字符算术运算经常被使用，因为它可以产生一些美观的代码。

每个字符变量分配一个字节。这意味着字符类型只能表示256个字符（包括控制以及非打印字符）。对于英语来说，这已绰绰有余了，但对于其他语言就要制订它们自己的非ASCII字符集。Unicode字符集努力在这方面进行规范化，以便使英语、法语、俄语、汉语和日语都适用于同样的16位字符集。

C++通过提供一个可扩展的字符集的宽字符类型wchar_t来支持上述的要求。这一类型

所分配的存储空间是基于某个整数类型的，比如int或short int。为了表示宽字符的字面值，用L（必须是大写）作为前缀。

```
wchar_t wc = L'a';
```

C++允许程序员使用任何字符集。为了实现可移植性，ASCII码可能是最佳选择。

3.2.3 布尔值

大多数现代的程序设计语言都支持取值为true或false的布尔值。布尔值在计算逻辑表达式、在从程序代码中选择执行路径等方面很有用。

C不支持布尔类型，而是把任何非零值看做是true，把零值看做是false。C允许在这些值上进行逻辑运算。（重复一下，任何非零的结果解释为true，而结果为零的解释为false）。对于实现逻辑运算来说，这已经足够了。今后将会看到，这种做法会造成编译程序无法识别的错误。开始，C++继承了这种做法，然而新的标准引入了两个值：true和false的bool类型，试图从某种程度上改变上述情况。

```
bool flag = false, result = true;
```

之所以说“某种程度的改变”而不是“改变”，是因为bool类型的使用并没有使C++完全消除从C那里继承而来的易出错的特征。原有的做法仍被认为是合法的，布尔值仍被看做是一个小整数。如果想打印上面定义的flag和result的值，那么第一个将被打印为0（不是false），而第二个将被打印为1（不是true）。

布尔值只占用一个字节的内存空间。由于布尔类型只有两个不同的数值，布尔类型的变量只需占用比一个字节更小的内存。1位就已经足够；也就是说，编译程序可以把8个布尔变量的值压缩在一个字节里。然而，这就需要有另外的代码对这些值进行压缩和解压。因为现代的计算机不能直接访问一个小于1个字节的内存单元。实际上，很多计算机都不能直接访问小于两个字节的内存单元。就运行速度而言，当以4个字节为单位访问内存时，计算机的存取速度最快。因此在很多现代的计算机上的整数都分配4个字节。

当开始使用关系运算符和逻辑运算符时，就会用到更多的布尔值。

3.3 浮点类型

整数和字符都属于整数类型：这些类型的值都是由1和0的不同组合构成的，它们不能包含小数部分。或者说，不管它们是什么样的位模式，C++编译程序都不能把它们解释为小数。如果需要用到小数，就必须使用其他类型。

C++没有给程序员提供控制在小数点之后的数字个数的定点数类型。但是，程序员可以使用由一个尾数（有整数部分和小数部分）和一个指数组成的浮点数。指数表示的是10的幂，在计算机的内存中，它当然被看做是2的幂。

在C++中有三个浮点类型：float、double和long double。但没有short double类型，它由float类型所取代。这些类型所占内存的大小依赖于机器。通常来说，float类型占用4个字节，double类型占用8个字节，long double类型则占用10个字节（或者和double一样，只占用8个字节）。

可以用浮点数表示尾数，其位数也是依赖于机器的。通常，float类型的数值有7位数字，double类型的数值有15位数字，而long double类型的数值有19位数字。

数值的范围依赖于该数值的指数部分所分配的位数。对float类型的数值来说，指数的取值范围是-38~38。对double类型的数值来说，其指数的取值范围是-611~611。对long double的数值来说，指数范围是-4932~4932，因此，用它来表示那些很大的数据已经是绰绰有余了。

下面是定义浮点类型变量的例子：

```
float pi;  double r;  long double d;
```

对浮点类型分层的目的与整数类型是一样的：给程序员在所占的内存大小与数值的精度和规模之间一个做出折中选择的机会。对于那些计算精度要求不高而空间紧缺的应用（大多数的实时嵌入式系统）场合，可以使用float类型。对于那些计算精度要求很高的应用（航空航海方面）场合，可以使用long double类型，尽管这些值占用更多的内存空间并且运算速度会变慢。至于所有其他的场合，使用double类型就可以了。

对于大多数的应用而言，从时间和空间来说，float类型会显得太短，而long double类型会显得太浪费。除非有特别的规定，使用double类型会比较合适。C++的math库中的所有函数都要求参数类型为double，并且返回double类型的数值（比如第2章中第一个C++程序中所用的函数pow()）。

浮点数据类型有固定的精度。在double类型中，很大和很小的数在给定的平台上都有着相同位数。如前面所述，C++没有提供小数点固定的数据类型（在小数点后有固定的位数）。

浮点类型的数值可以采用基数表示法（有小数点）或者科学计数表示法（用E或e表示指数）。

```
double r=5.3; long double d=530.0e-2;
```

这两个数表示了同一个数值。10^{-2}等于1除以10^2，也就是，1除以100。

在这个例子中，科学计数表示法并不比一般的基数表示法更优越，但要简洁地表示非常大或非常小的数值时，用科学计数表示法就会很方便。

大多数浮点数字面值的尾数都有3个组成部分：整数部分、小数点（当然只有一个）和小数部分。并不是所有这三个组成部分在所有情况下都是不可缺少的。

然而，使用某一种表示法来区分浮点数和整数是重要的。因此，既有小数点又有小数部分而没有整数部分是允许的；而既有整数部分又有小数点而没有小数部分也是允许的。

```
double small = .09, large = 5.;
```

甚至同时没有小数点和小数部分但有指数部分的形式，也能表示这是一个有别于整数的浮点数。

```
double big = 500e2;
```

在科学计数表示法中，指数必须是一个整数，尽管在数学上指数可以是任意形式的数。C++只接受整数形式的指数，另外指数是正数时还可以选择是否加上符号。

```
double big = 500e+2;              // big = 500e+2.2; is no good
```

和整型数值类似，C++允许对数值加上修饰符来区分不同类型的数值。修饰符'f'或'F'表示的是float浮点数，修饰符'l'或'L'表示的则是long double类型的数值。于是我们也许会推断出修饰符'd'或'D'表示的是double浮点数，但这是一个不正确的推断。所有没有修饰符的浮点数字面值缺省为double数值。

```
float pi = 3.14f;  double r = 5.3;  long double d = 5.3 L;
```

3.4 C++表达式的使用

表达式由操作数和运算符组成。操作数可以是任何具有类型的值；即可以是变量、函数返回值或者另一个由操作数和运算符组成的表达式。运算符是一些在C++中已有指定意义的符号。把运算符作用于操作数便可产生一个可用于其他表达式的值。使用空格对提高可读性有帮助，但并不是必需的。

```
x = (a + b) * (a + 2*b) * (a+3*b);    // space is optional
```

运算符有两个属性影响了表达式求值的顺序，它们是：运算符的优先级（优先级高的运算符先执行）和运算符的结合性（决定同一优先级的运算符是从左向右还是从右向左求值）。

在C++中有56个运算符，我们需要非常努力地学会其中的18个优先级。表3-1列出了C++的运算符。很显然，想要通过阅读就掌握这个表格是不可能的。把它列在这里的目的只是提供参考，而不是为了让你记住。我们将会通过使用这些运算符而逐渐掌握它。当我们不太确定运算符的优先级时，可以使用圆括号来指定。毕竟，即使我们把所有运算符的优先级都熟记于心，今后的维护人员也许对此表掌握得并不太好，因此也会混淆求值的顺序。

表3-1　C++运算符

类　别	运 算 符	结 合 性
作用域符号	::	从左向右
最重要的	() [] -> typeid dynamic_cast static_cast reinterpret_cast const_cast	从左向右
次要的	++ -- ~ ! + - * & Sizeof new delete (type)	从右向左
成员选择	. -> * .*	从左向右
乘除运算	* / %	从左向右
加减运算	+ -	从左向右
移位运算	<< >>	从左向右
关系运算	< <= > >=	从左向右
等于运算	== !=	从左向右
按位与运算	&	从左向右
按位异或运算	^	从左向右
按位或运算	\|	从左向右
逻辑与运算	&&	从左向右
逻辑或运算	\|\|	从左向右
赋值运算	= *= /= %= += -= <<= >>= &= != ^=	从右向左
条件运算	? :	从右向左
抛掷符	throw	
逗号运算	,	从左向右

3.4.1 高优先级运算符

在表最上面的最高优先级处，可以看到那些比其他的运算符更能紧密地绑定操作数（这是最高优先级的另一种说法）的运算符。例如，圆括号是高优先级的运算符。不管在表达式中使用了什么运算符，圆括号里的子表达式将最先被求值。

现在可以讨论的另外两个运算符是正号'+'和负号'-'运算符。作为一元运算符，它们只

有一个操作数。例如，+2.0，-2.0。也许有人会问，一元的正负号运算符和我们所熟悉的低优先级的加减法运算符之间有什么不同呢？其区别是：一元的正负号运算符只有一个操作数，而加减法运算符有两个操作数。这一区别使得我们不需用额外的圆括号就可以书写带一元运算符的复合表达式。例如，2.5- -0.25。由于一元负号是一个运算符，它与其操作数之间可以有任意数目的空格，例如，可将上式写为2.5- -0.25，这不会影响到表达式的求值。当然，如果想使之容易为维护人员所理解，最好将表达式写成2.5-(-0.25)。

本章前面曾讨论过sizeof运算符。这是C++中惟一一个能够同时对类型标识符和变量名字进行操作的运算符。

```
int x = sizeof(int);  int y = sizeof(x);      // same values
```

在这里，x得到整数所分配的字节数，而y得到变量x（它刚好是一个整数）所分配的字节数。和任何一元运算符一样，当操作数是某一个变量的名字时，sizeof的操作数可以不用圆括号括住。

```
int x = sizeof(int);  int y = sizeof x;       // same results
```

大家或许会认为，如果操作数是类型名的话，情况也会一样，但情况却不是这样的。这时必须有圆括号。

```
int x = sizeof int;  int y = sizeof x;            // not OK
```

关于高优先级运算符暂且只能讨论到这里，不久，将会有更多的讨论。

对于下一个优先级的成员选择运算符以及接近表尾的throw运算符来说，也有同样的情况。类型运算符（类型转换）将在本章稍后进行讨论。

3.4.2 算术运算符

在表3-1的第5、6个优先级中，有乘法和加法运算符。在此可以对它们简单地讨论一下。乘法及除法的优先级高于加法及减法。当需要改变求值顺序时，就要使用圆括号。

```
x = (a + b) * (a + 2*b) * (a+3*b);    // buyers beware
```

除非结果有溢出，C++是不会产生任何异常的。因此，程序员的责任就是要保证无论程序处理什么输入数据，都不会产生溢出。

在整数和浮点数上都可以进行算术操作。尽管除法运算符'/'也适用于整数和浮点数，但作用是不同的：作用在浮点数上时，其结果是一个以某个精度计算所得的浮点数；作用在整数上时，其结果是截去余数后的一个整数。例如，7/3对浮点操作数来说，结果是2.333333，对整数操作数来说则是2。

取模运算符'%'返回整数除法的余数，它只适用于整数类型（整数、字符），不能用于浮点数。例如，7除以3的余数为1，因此7模3等于1。类似地，8除以3的余数为2，因此8模3等于2。由于9除以3得3而无余数，因此9模3等于0。

```
int x1=7, x2=8, x3=9;  int r1, r2, r3;
r1 = x1 % 3; r2 = x2 % 3;  r3 = x3 % 3;          // r1 is 1, r2 is 2, r3 is 0
```

当第一个操作数小于第二个操作数时，情况是一样的。例如，5除以7得0余5，因此5模7等于5。类似地，6除以7得0余6，因此6模7等于6。由于7除以7得1而无余数，因此7模7等于0。

```
int a1=5, a2=6, a3=7;  int r1, r2, r3;
```

```
r1 = a1 % 7; r2 = a2 % 7;  r3 = a3 % 7;          // r1 is 5, r2 is 6, r3 is 0
```

对于操作数为正数的情形，这是相当直截了当的。如果是负操作数，那么其结果则依赖于机器。所幸的是，从来不需要对负数使用取模运算符。这个运算符通常用于判断容器尾部是否还有空闲空间，或者是否已被数据填满了（这就得返回到容器的开头），而容器的长度以及容器中的下一个位置永远不会为负数。

从左向右的结合性表示，当同一个表达式中使用了几个优先级相同的运算符时，它们按照从左向右的次序进行求值。这对乘法和加法并不重要，但对减法和除法却非常重要。不管我们把a+b+c的求值理解为(a+b)+c或a+(b+c)，结果都是一样的。重要的是应该把a-b-c的求值理解为(a-b)-c而不是a-(b-c)。类似地，a/b/c意味着(a/b)/c而不是a/(b/c)。

加1运算符'++'和减1运算符'--'是C/C++程序设计特有的。它们是只带一个操作数为1的加法和减法运算符，因此它们只需要指定一个操作数。它们实现了汇编语言类型的处理：以不可中断的高优先级进行加1或减1。从某种意义上说，这些运算符对其操作数会产生副作用。

```
int x = 6, y = 10;  x++;  y--;          // now x is 7, y is 9
```

这些运算符的基本形式很简单。加1运算符将操作数加1，减1运算符将操作数减1。这个例子完全等同于下面的代码：

```
int x = 6, y = 10;  x = x + 1;  y = y - 1;       // now x is 7, y is 9
```

那些从使用其他非C语言转到使用C++语言的程序员经常会感到困惑：既然它们完全等同于一般的加减法，为什么要使用加1和减1运算符呢？以前的答案是：编译程序为加1和减1运算符产生目标代码的效率，比为一般的加法和减法产生目标代码的效率要高。但现在这个答案已经不再正确了。使用了现代的编译程序设计技术以及目标代码优化技术，两种方法的性能已没有什么差别。

今天，这只是一个风格问题，当然，并非一定要使用加1和减1运算符。可以使用一般的加法和减法，而且程序将和使用加1和减1运算符的程序一样优雅、正确和快速。只是老板（以及很可能同事）可能会怀疑你是否是一个真正的C++方面的专家而已。

实际上，加1和减1运算符是有相当多的用途的。它们的使用不局限于整数。也允许浮点数使用它们。

```
float small = 0.09;  small++;          // now small is 1.09
```

加1和减1运算符有两种类型：前缀和后缀运算符。前面的例子中用到的是后缀运算符：运算符跟在要修改的操作数之后。前缀运算符则位于其操作数的前面。下面是一个使用前缀运算符的例子：

```
int x = 6, y = 10;  ++x;  --y;          // now x is 7, y is 9
```

那么这两种形式的差别是什么呢？因为这里的前缀运算符的结果好像和后缀运算符的结果一样。确实，在这个环境中前缀运算符的作用等价于下面的代码：

```
int x = 6, y = 10;  x = x + 1;  y = y - 1;          // now x is 7, y is 9
```

可见，结果与前面的完全一样。前缀和后缀运算符之间的差别是它们在表达式中如何使用。这些操作的结果是一个数值（和C++中所有其他的操作一样，这是一个重要的原则）。在

我们的例子中，x++和++x的返回值都是7，y--和--y的返回值都是9。这些值可以用在所有可接受整数值的表达式中。只是在这些表达式中前缀和后缀运算符的行为会有所不同。

当使用前缀运算符时，操作数的值首先被加1（或被减1），然后其结果才被用在表达式中。

```
int x=6, y=10, a, b;  a = 5 + ++x; b = 5 + --y;        // a is 12, b is 14
```

请注意前缀运算符前的空格，使用它们可以避免混淆。编译程序（及维护人员）可能会对5+++x感到困惑，尽管编译程序和维护人员都不会混淆5+--y。

当使用后缀运算符时，操作数的数值首先用于表达式中，然后再对该变量加1或减1。

```
int x=6, y=10, a, b;  a = 5 + x++; b = 5 + y--;        // a is 11, b is 15
```

大家可能觉得使用加1和减1运算符编写的代码是很容易混淆的。可能如此。然而，这些运算符由于它们简单的形式而非常流行：比如程序3-1（或这一章前面一些其他的例子）的循环体，在每一次重复地对其计数器或下标进行加1或减1时，没经验的C++程序员将不会用加1运算符来编写这段代码（见程序3-5；当然其输出的结果和程序3-1的结果一样）。

程序3-5 加1运算符例子

```
#include <limits>
#include <iostream>
using namespace std;
int main(void)
{
  int num = INT_MAX - 2;
  int cnt = 0;
  cout << "Integer overflow in C++:" << endl;
  cout << "Incrementing from " << num << endl;
  while (cnt < 5)
  { num++;  cnt++;                          // increment operators
    cout << cnt << "    " << num << endl; }
  cout << "Thank you for worrying about integer limits" << endl;
  return 0;
    }
```

程序员可能很快就会喜欢使用加1和减1运算符。如果现在对使用它们感到不习惯，也可以使用与其他语言一样的算术运算符。然而，如果总是对加1和减1运算符避而不用，老板可能会怀疑你不能流利地使用C++。注意随时使自己的行为与别人保持一致。

为了便于大家阅读，我们可以跳过接下来的3.4.3节“移位运算符”和3.4.4节“按位逻辑运算符”的内容。

3.4.3 移位运算符

在C++运算符表中，接下来的运算符就是移位运算符‘<<’和‘>>’。不过，这两个不是移位运算符！而是与输出对象cout以及输入对象cin一起使用的插入和抽取运算符。在这里使用的是一种叫做运算符重载（operator overloading）的设计技术。很早以前，C语言中就有了移位运算符。在设计C++语言时，设计者决定把已有的运算符应用于新的环境中。也就是说，我们是在学习一个已有运算符的新含义，而不是在学习新的运算符（或新的关键字）。

实际上，重载运算符技术并不真是一种新技术。例如，在C++中运算符'+'有着多少种含义呢？a) 作为一元加运算符，b) 用来进行整数相加，c) 用来进行浮点数相加，(并且这些操作的

实现彼此不同,也不同于加法)，d) 作为加1前缀和后缀运算符的一部分。目前我们还没有结束对运算符的讨论。

移位运算符把其第一个操作数的值向左或向右移动，第二个操作数指定了第一个操作数要移动的位数。实际上的操作并不像听起来的那么复杂。先考虑右移的情形。

```
int x=5, y=1, result;  result = x >> y;       // result is 2
```

该运算符会向右移动其第一个操作数的位模式（在本例中第一个操作数是值为5的x），移位的位数由第二个操作数（本例中第二个操作数是等于1的y）指定。整数5的二进制表示是101。当我们把这个位模式向右移一位时，便得到位模式10，它对应于整数2。在这种情况下，右移是一种用2（或由第二个操作数指定）去除整数的一种快捷方式。

左移运算符向相反的方向移动位模式。这里，位模式101变为1010，它对应于十进数10。

```
int x=5, y=1, result;  result = x << y;       // result is 10
```

当位模式向左移动时，从第一个操作数中移出的位将丢失；操作数右边的位被补上了0（与前一个例子一样）。类似地，当位模式向右移动时，移出去的位也将丢失，而操作数左边取什么值是依赖于机器的。

有符号整数的最左一位是符号位。如果它是0，表示该值为正；如果是1，表示该数值为负。当它是正数时，没有什么问题：符号位的0向右移，而且0从左移到符号位。如果该数值是负的，符号位的1向右移，此时便可能会产生可移植性问题。在某些计算机上，1被移到符号位上（以及在有必要时继续前移），这种情形称为算术位移。在另外的一些计算机上，0被移到符号位（以及有必要时继续右移），这种情形称为逻辑位移。

3.4.4 按位逻辑运算符

按位逻辑运算符包括按位与运算符‘&’、按位异或运算符‘^’、按位或运算符‘|’和更高优先级的按位求补（反）运算符‘~’。前面三个运算符都是二元的，最后一个运算符是一元运算符（它只需要一个操作数）。

和移位运算符类似，逻辑运算符作用在位模式上。它们对两个操作数的每一个位进行运算，而操作数上两个对应位的单独运算就得到结果的对应位。

如果两个操作数的对应位均为1，按位与运算符的结果位就为1；如果至少有一个（或两个）操作数的对应位为0，其结果位就为0。在以下的例子中，只考虑操作数有4位，以及其结果的所有其他位都设置为0的情形。为了说明按位与运算符，假定第一个操作数是12（其二进制编码是1100）以及第二个操作数是10（其二进制编码是1010）。将第一个操作数和第二个操作数的对应位进行比较，可以看到只有首位上的两个位才同为1，而在其他位上不是一个位为0就是两个位均为0。因此，运算的结果就是1000（十进数8）：1100&1010等于1000。

如果一个或两个操作数的对应位是1，按位或运算符的结果位就为1；只有两个操作数的对应位均为0，其结果位才为0。就操作数12（二进数1100）和10（二进数1010）而言，其结果位除了最右一位之外，其他位均为1，于是得到二进制编码1110（十进制数14）：1100 | 1010等于1110。

如果只有一个操作数的对应位是1，按位异或运算符的结果就为1；如果两个操作数的对应位相同（均为0或均为1），其结果位就为0。在我们的例子中，两个操作数的最左位和最右位都一样，中间的两个位不同，于是得到的结果为二进制编码的0110（十进制数6）：

1100^1010等于0110。

按位取反运算符把结果位设置为与操作数的对应位相反。如果操作数位是1，结果位就是0；如果操作数位是0，结果位就是1。比如，对12（二进制数1100）求补运算的结果为二进制编码0011（十进制数3）、也就是：~1100等于0011。

这些运算符经常用在处理大量状态信息的应用中：通信频道的开通或关闭、设备是否准备就绪、电线是高压还是低压、顾客是否有好的信用，于是可以给予较高的折扣等等。在一个大型的机器上，我们可以为每一个这样的值分配一个整数大小的空间，即使只使用了其中的一位，使其值为0或1。在一个较小型的机器上，可以使用一组布尔值，为每一个这样的布尔值分配一个字节。而在小型的机器上，以上两种做法都是浪费的。因此经常把这类信息压缩到状态字里，使得位模式中的每一位（标记位）都具有它自己的含义。为了从状态字中抽取某一个特定位的值或者设置某个位的值，可以对特定的位模式（掩码）和常量使用移位和逻辑运算符。

比如在状态字（例如它的名字是flags）中，从右数起的第三个位代表着一台设备的开关状态，则当设备开始运作时，程序应把该位设置为1。

为了能够把该位设置为1，应该设置一个其第三个位为1的变量（例如命名为onMask）。如果让变量flags和onMask的第三位使用按位或运算符，则不管flags的第三位原来的状态如何，运算后flags的第三位将置为1。这比先检查flags的第三位是什么然后决定是否要使用按位或运算符要快。注意不能对单独一个位使用逻辑运算符，它们必须同时作用于操作数的所有位，这意味着变量onMask的其他所有位（第三个位除外）应具有不改变变量flags其他位的值。就按位或运算符而言，这些值是0。

以下就是如何设置掩码以便处理压缩位的方法：把保证所需状态的位设置为所要求的值，并把其他位设置为不改变已有状态的值。在上述的例子中，应该使变量onMask第三位设置为1，而将其他所有位设为0。如果是一个4位数的形式，onMask的位模式就是0100或十进制数4。

```
int flags, onMask = 4;
flags = flags | onMask;        // this sets the 3rd bit to 1
```

当要关闭设备时，就要将第三位重置为0，而其他位不变。需要有使第三位设置为0的另一个掩码（例如命名为offMask），它与任何值进行按位与运算都会产生0。为了使其他各位不变，应该把掩码的其他位设置为1。因此，变量offMask应为位模式的1011或十进制数11。

然而，这里还有一个问题。只有当位数为4位时，位模式的值才等于11。对于位数为8位而言，位模式应为11111011，对应的十进制数值就等于244。如果有16位或32位，就还需要其他的位模式。这是可移植性问题的一个典型的例子。这里的解决方案很简单。所有这些位模式都是位模式0100的不同长度的反码。因此，实现初始化变量offMask的可移植方法是使用与位模式0100相反的位模式：

```
int offMask = ~onMask;
flags = flags & offMask;              // this resets the 3rd bit to 0
```

为了检查第三位是否为1，可以将掩码onMask与变量flags进行按位与运算。这个操作会将结果的所有位都设置为0，但第三位除外（因为除了第三位外，onMask的其他位均为0）。如果flags中的第三位是0（表示设备关闭），结果就是0（false）。如果flag的第三位为1（设备开动），结果就是非0（true）。另一种存取位值的方法是把flags的位模式向右移2位，然后将它与除了最右位之外其他位均为0的掩码进行与操作。如果结果是1，说明该位是1；如

果结果是0，说明该位为0（我们将很快讨论相等运算符）：

```
if (((flags >> 2)&1)==1) cout << "3rd bit ON\n";        // test it
```

如果我们不需要开发嵌入式系统和通信系统，就不会经常对值进行移位等操作。可以对此部分内容少花点精力；否则，就必须多做这方面的练习，因为在这些系统中，移位和逻辑运算是经常会使用到的。

3.4.5 关系和相等运算符

关系运算符在所有的应用中都会用到。C++支持4个关系运算符：小于‘<’、小于或等于‘<=’、大于‘>’以及大于或等于‘>=’。和C++中其他的双字符运算符一样，双字符运算符中的两个符号必须紧靠在一起书写。这些运算符主要用于条件语句和循环语句中进行比较操作。例如，在程序3-1中，循环条件检查是否满足cnt<5。如果满足（在第一次循环时），就执行循环体。如果cnt的值增大到5，那么条件5<5不真，循环就结束。这看起来很简单，但这是从C那里继承下来的，并不像它看起来那样简单。

C++没有提供独立于整型的基本布尔类型。前面讨论的布尔类型是用一个小整数来实现的：true用1表示而false用0表示。这就是说，C++中的比较运算结果不是true或false（和其他所有程序设计语言不一样），而是1和0这两个数。这个整数只有1个字节的大小，但如果有必要，则可以用更大的整数来表示。

因此，如果x大于y，则x>y的值就为1；否则，其值就为0。如果x小于y，则x<y的值就为1；否则，其值就是0。类似地，如果x不小于y，则x>=y的值为1；如果x小于y，则其值为0。如果x不大于y，则x<=y的值为1；如果x大于y，则其值为0。

这并不会改变比较的形式和它们的工作方式，但把逻辑值当做数来使用就很可能会造成滥用。比如，x>y>z的值是多少？在大多数程序设计语言中（虽然有某些例外），这是一个语法错误。

但在C++语言中，它是一个完全合法的表达式。由于关系运算符从左向右的结合性，首先比较x和y。如果x大于y，结果就是1；接着比较1和z。如果1大于z，则表达式的值就为1；否则就是0。然而，如果x不大于y，则结果就是0；接着就比较0和z。如果0大于z，则表达式的值为1；否则就是0。这样的不等式很难理解。

在运算符表格的后面有相等运算符。C++支持两个相等运算符：相等运算符‘==’和不等运算符‘!=’。同样，这些运算符中的符号不能分开书写。当比较运算的值为真时，运算符返回1；当比较运算的值为假时，运算符返回0。

所以，如果x等于y，则x==y的值为1；否则为0。如果x不等于y，则x!=y的值为1；否则为0。

假定想在x等于y时把z的值设为10，在x不等于y时把z的值设为9。在所有程序设计语言（包括C和C++）中，可以直接写为：

```
if (x == y)          // set z to 10, or to 9
  z = 10;
else
  z = 9;
```

使用C++我们就可以简单地写为：

```
z = 9 + (x == y);
```

它比前一种形式更难理解，但很明显它显得更加简洁和优雅。

由于所有的值，不仅仅是1，都可以当做true来使用，于是这会使得情况变得更坏。另外，所有的运算都会返回一个值，包括赋值运算符。例如，以下的赋值语句把y的值赋给变量x，并返回该值，今后如果有需要，返回值可用于表达式中。

```
x = y;
```

这意味着，如果意外地把相等运算符'=='误写为赋值运算符'='，就只能依靠自己才能发现错误。这种相等运算符的误写并不会导致语法错误（虽然它应该是），它完全是一个合法的表达式。

例如，假设在上述例子中x和y的值均为1。这就意味着表达式应该把z的值设为10。现在假设我们误写了第一个表达式：

```
if (x = y)
  z = 10;
else
  z = 9;
```

这一语句把x的值设为y（它并没有改变x的值，因为在本例中x和y有着相同的值1），并把这个值返回给if语句，它将条件解释为真（因为条件的值不是0），于是把z设为10。运行结果符合预期。

由于测试次数很少，程序员会很容易相信该程序运行是正确的。如果多用一组不同的值来测试，比如x是1和y是2，就会发现z的值是10而不是9（同样，这里的赋值运算x=y将返回2；这就使条件解释为true，因为条件的值不是0）。

现在，假设将第二个表达式错写为：

```
z = 9 + (x = y);
```

该赋值运算符返回x变为y值之后的值再加上9，从而设置z的值。变量z的值将既不是9也不是10，而是11。

如果只是第一次看到这种情况，有人也许会认为这没什么大不了的，因为赋值运算符'='和相等运算符'=='之间的差别并不是小到难以相互区分。当然，这里讨论的目的并不是关于它们区别的大小，只是要说明，无论如何，我们经常会将'=='错写为'='。这种错误的积累将使软件业浪费非常多的时间、精力和金钱去搜寻这些错误。

当使用赋值运算和比较运算时，请一定要检查是否正确地拼写了它们！我们一定要重视这个问题。

警告 将相等运算符'=='误写为赋值运算符'='不是一个语法错误。其结果会产生一个合法的C++表达式，编译程序会无警告地产生错误的代码。记住要检查条件表达式中是否存在这样的错误。

3.4.6 逻辑运算符

下一组运算符包括以下的逻辑运算符：逻辑与运算符'&&'、逻辑或运算符'||'和逻辑非运算符'!'。与按位运算符类似，与和或运算符是二元运算符（它们要求有两个操作数），逻辑非运算符是一元运算符。在逻辑运算符中没有提供异或运算。

逻辑运算符很常用，它们与关系运算符一样重要，很难找到一个没有使用过这些运算符的程序。为什么这些运算符的形式是由按位运算符派生出来的呢？因为这些运算符是C++从C那里继承下来的，它们仅次于按位运算符。

与按位运算符不一样，逻辑运算符把每一个操作数当做一个整体。如果操作数的值为0，则认为它是false；如果操作数的值非0，则不管它是什么，都会被认为是true。

逻辑与运算符‘&&’只有在其操作数均为非0的情况下才返回1（所占的空间与bool类型一样）；否则返回0：

```
if (x < y && y < z) cout << "y is between x and z\n";
```

逻辑或运算符‘||’在其中一个操作数非0时返回1；只有两个操作数都为0时返回0：

```
if (x > 0 || y > 0) cout << "At least one is positive\n";
```

逻辑非运算符‘!’在其操作数非0时返回0；在其操作数为0时，则返回1。总是可以通过适当地修改其他的条件来避免使用这个运算符。有时，使用非运算符会更简单。例如，考虑这样的一个程序：它能够给有良好信用度（rating==2）的老年市民（age>=65）提供打折优惠。对没有资格享受打折的人，对条件进行否定并不难，但程序员可能会发现这样写会更容易一些：

```
if (!(age >= 65 && rating == 2)) cout << 'No discount\n";
```

整数和浮点数对象均可作为逻辑操作数：任何非0值可作为true，而任何0值可作为false。注意，没有必要把逻辑操作的操作数置于圆括号中。然而，if语句（和while语句）的逻辑表达式必须写在圆括号中。因此上一个例子中有两对圆括号。

和其他语言相同的是，逻辑运算符按从左向右的顺序求值；和其他语言不同的是，‘&&’运算符的优先级高于‘||’运算符。于是在书写复杂的逻辑表达式时可以不必使用圆括号。例如，要提供10%的折扣给信用度为2的老年市民以及有200美元或更大数目订单的新顾客，就可以表示为下面的形式：

```
if (age>=65 && rating==2 || first_time == true && total_order>200.0) discount = 0.1;
```

这样书写复杂的表达式并不总是最好的方法。在一个复杂的表达式中，一种好的书写方法是使用圆括号来向维护人员表明该表达式的组成是什么。

```
if ((age>=65 && rating==2) || (first_time == true && total_order>200.0)) discount = 0.1;
```

在这个例子中，子表达式两旁的圆括号是可选的。有时它们是必须的。例如，如果信用度为1或2的老年市民都有资格享受打折优惠，那么没有圆括号的逻辑表达式将是不正确的：

```
if (age>=65 && rating==1 || rating==2) discount = 0.1;  // incorrect logical expression
```

这个语句会把打折优惠提供给信誉值为1的老年市民和所有信誉值为2的顾客，而不只是提供给老年市民（记住，与运算符‘&&’的优先级高于或运算符‘||’）。圆括号的使用改正了这个问题。

```
if (age>=65 && (rating==1 || rating==2)) discount = 0.1;   // correct logical expression
```

注意　逻辑与运算符的优先级高于逻辑或运算符，可以使用圆括号来帮助维护人员理解复杂的逻辑表达式的含义。

C++逻辑运算符是一种“短路”（short circuit）运算符。这意味着它先对第一个操作数求

值，并且如果第一次计算就可以确定表达式的结果的话，就不再对第二个操作数求值。在下一个例子中，如果x不小于y，那么再去检验y是否小于z就变得没有什么意义：将不能得出y位于x和z之间的结论；因此在这种情况下，不会对第二个条件求值。

```
if (x < y && y < z) cout << "y is between x and z\n";
```

在本例中，可以节省几微秒的时间；但这并不重要，不久，将会讨论一个通过这种特性能够维护代码完整性的例子。

3.4.7 赋值运算符

赋值运算符（及其变种）的优先级较低。这种规定是合理的，因为它必须在表达式中其他所有的操作都执行完之后才能执行。将赋值定义为一个运算符既在语法上是令人鼓舞的突破，又会导致一些危险。在内存中任何有地址的操作数都可以作为赋值的目标，而且该值还可以直接用于其他表达式中。

与内存地址相关的术语是*左值*（lvalue），它意味着表达式可以用在赋值运算符的左边。它具有一个地址，并且当它作为赋值运算的目标时，这个地址上的值将被修改。到目前为止我们只见过一种左值：变量的名字。C++中还有其他的左值，我们将在今后对它们进行讨论。注意，一个左值也可以用于赋值运算符的右边。

另一种C++值是*右值*（rvalue），它具有一个值，但它在内存里没有地址，可以让程序修改这个值。右值的例子是值、函数的返回值、二元运算的结果。右值只能用作赋值运算符右边的表达式，而不能用作赋值运算的目标。下面是把右值用作左值的错误例子。它们都被标记为语法错误。

```
5 = foo();              // a literal should not be used as an lvalue
foo() = 5;              // return value should not be used as an lvalue
score * 2 = 5;          // result of operation should not be used as an lvalue
```

和其他的语言不同，C++赋值运算是一个可以使用右值的二元运算符。这就允许链式赋值：

```
int x, y, z;    x = y = z = 0;
```

赋值运算从右向左结合：x=y=z=0；表示x=(y=(z=0))；而不是((x=y)=z)=0；由于x=y不是左值，因此不能对它赋值。这一特征易被滥用

```
x = (a = b*c)*4;     // this is legal in C/C++
x = a = b*c*4;       // this has a different meaning
x = 4*a = b*c;       // syntax error: there is no lvalue for 4*a
```

除了传统的赋值运算符之外，C++还有很多赋值运算符变种——算术赋值运算符，其目的是缩短算术表达式的长度。例如，可以用x+=y；代替x=x+y；其结果是一样的。赋值运算符可以用于所有二元运算符（'+=' '-=' '*=' '/=' '%=' '&=' '|=' '^=' '<<=' 和 '>>='）中。它们几乎和加1、减1运算符一样流行，其使用目的也是一样的。下面是一个计算前100个整数的平方和的代码段：

```
double sum = 0.0;  int i = 0;
while (i++ < 100)
  sum += i*i;  // arithmetic assignment
cout << "The sum of first 100 numbers is " << sum << endl;
```

下面是用一些传统的运算符书写的、完成同样功能的代码段：

```
double sum = 0.0;  int i = 0;
while (i < 100)
  { i = i + 1;
    sum = sum + i*i; }
cout << "The sum of first 100 numbers is " << sum << endl;
```

以前曾经提过，编译程序在两种情况下所产生的目标代码都是一样的，其区别只是美观性而已。每一个C++程序员必须学会欣赏这种速记运算符的表示能力。

3.4.8 条件运算符

C++的运算符中还有一个运算符就是条件运算符。它是C++中惟一的一个三元运算符，它有3个操作数。运算符自身含有两个符号：'?'和':'，但与其他双符号运算符不同，这两个符号由第二个操作数分隔。下面是条件运算符一般的语法形式：

```
operand1 ? operand2 : operand3      // evaluate operand2 if operand1 is true
```

在这里，operand1是测试表达式，它可以是任何的标准类型（简单类型没有程序可存取的成分），包括float类型。这个操作数总是被最先求值。如果第一个操作数的求值结果为true（非0），那么就对operand2求值，而跳过operand3。如果第一个操作数的求值结果为false（0），那么跳过operand2，而对operand3求值。为了下一步的使用而返回的值是operand2的值或是operand3的值；该选择是在operand1的值的基础上做出的。

不要被叙述中所用的true和false所误导。表达式operand1当然可以是一个布尔表达式，但不一定要如此。C++允许使用任何一个可取得0和非0值的类型。

在下一个例子中，把变量y和变量z中的最小值赋值给变量a。在这里，operand1是表达式y<z；如果这个表达式为true，operand2（此时是变量y）就被求值，且将它的值作为表达式的值返回；如果表达式y<z为false，就返回operand3（此时是变量z）的值。刚好它又是z的值，但这没有关系。

```
a = y < z ? y : z;       // a is set to minimum of y, z
```

注意，这里不必像其他使用逻辑表达式的所有情形那样，把operand1置于圆括号中。（如果使用圆括号，它可能更加有利于阅读。）条件运算符简洁而优雅，但可能难于读懂，特别是当其结果还要用在其他表达式中时。在下面的例子中，使用if语句也可以达到同样目的：

```
if (y < z)
  a = y;                    // a is set to minimum of y, z
else
  a = z;
```

下面是另一个展示了条件运算符优点的例子，其返回值被作为另一个表达式（输出语句）的一部分：如果申请者的分数高于80，该语句就打印"Your application has been approved."；否则，它打印"Your application has not been approved."。

```
cout << "Your application has" << (score > 80 ? "" : " not")
     << " been approved.\n";
```

用传统的方法编写会罗嗦一些，但可能更加易于读懂。

```
if (score > 80)
   cout << "Your application has been approved.\n";
else
```

```
cout << "Your application has not been approved.\n";
```

3.4.9 逗号运算符

其他语言并不会把逗号当做运算符，但C++却会这样。它把按照从左向右的次序求值的操作数连接在一起，并为今后的使用而返回最右那个表达式的值。如果需要在C++语法中只允许出现一个表达式的地方对几个表达式求值，那么使用逗号运算符就很方便。

```
expr1, expr2, expr3, ... , exprN
```

从最左边一个表达式开始，对每一个表达式求值；由于逗号的优先级最低，它在最后才执行；并会返回最后那个表达式的值。这被认为是最左表达式的副作用。下面是前面出现过的一个例子，在此去掉了块定界符。

```
double sum = 0.0;  int i = 0;
while (i < 100)
   i = i+1,  sum = sum + i*i; // no block delimiters are needed
cout << "The sum of first 100 numbers is " << sum << endl;
```

虽然这不是一个好方法，但把逗号当做运算符是合法的。这是一个故意滥用的例子。但这是无害的。当在无意中使用逗号运算符时，就会很危险。而由于代码不违反C++的语法规则，因此由不正确的代码导致的结果不会被认为有语法错误。例如，考虑用循环来计算平方和的第一个例子。

```
double sum = 0.0;  int i = 0;
while (i++ < 100)
  sum += i*i,  // arithmetic assignment
cout << "The sum of first 100 numbers is " << sum << endl;
```

第一个版本与这个版本惟一的不同是，在循环体结束处用逗号来代替了分号。不幸的是，这个做法并没有使代码从语法上出现错误。它可以通过编译并且运行，但会错误地运行。它会打印结果100次而不是一次。这还是一个容易发现的错误，但如果在循环后面的语句所做的工作不那么明显，错误将很难发现。当心逗号运算符，特别是在它以合法的C++运算符的身份出现在错误的场合时。

警告 由于逗号运算符是一个合法的C++运算符，错误地使用逗号时编译程序可能不会指出错误。

3.5 混合型表达式：隐藏的危险

C++是一种强类型语言。这就意味着如果上下文要求使用某种类型的值，那么使用另外一种类型的值来代替就是一个语法错误。这个重要的原则使得程序员不需要花费太多的努力就可以排除错误：编译程序会告诉程序员代码是不正确的，而不必让程序员通过艰苦的运行测试来发现错误。

例如，考察曾经在第2章中使用过的TimeOfDay类型。它是有两个整数成分的复合类型（不是一个标量类型）。存在设置其域值和存取它们的记号，而且这就是所允许的全部操作。不能对TimeOfDay变量加上2，或将它和另一个TimeOfDay变量进行比较。因此下面的代码段从语法上来说是不正确的：

```
TimeOfDay x, y;
x.setTime(20,15); y.setTime(22,40);      // this is OK: legitimate operations
x += 4;                                  // syntax error: incorrect operand type
if (x < y)                               // syntax error: incorrect operand type
   x = y - 1;                            // syntax error: incorrect operand type
```

然而，对于数值类型来说，C++是弱类型的。如果x和y都是int类型的话，则上述例子的最后三行从语法上来说将是正确的。另外，对于所有其他的数值类型：unsigned int、short、unsigned short、long、unsigned long、signed char、unsigned char、bool、float、double、long double来说，这三行语句也是正确的。另外，即使变量x和y分别属于不同的数值类型，这三行也是语法正确的。尽管这些变量占用不同大小的存储空间，并且对它们的位模式的解释也不同，但运行时这些操作都能正确地执行。

这与其他强类型的语言很不相同。例如，以下代码在C++中是可以接受的（也是常见的）：

```
double sum;
  . . .
sum = 1;                    // no syntax error
```

从现代强类型语言的观点来看，编译时会指出这是一个由程序员造成的不一致性的明显错误。在程序的某个地方，程序员说变量sum是double类型；而在另一个地方（这个地方和前一个地方可以被很多行代码隔开），程序员却把这个变量当做一个整数。如果这个代码被认为有语法错误，那么程序员就有机会去考虑它并决定怎样去消除这种不一致性：把变量sum定义为整数，或用下面的语句代替最后一行代码：

```
sum = 1.0;
```

在现代的强类型语言里，算术运算必须在两个类型完全相同的操作数上执行。对于一个C++程序员来说，会对这个问题有异议：多数人认为这个语句的两种版本都是可接受的，并不需要进行讨论。

当然，理想化的情形是按照强类型的原则，一个表达式中的所有操作数应该属于完全相同的类型。然而，对于数值类型，这一规则从某种程度上被放宽了。C++允许我们在同一个表达式中混合使用不同数值类型的值。

在目标代码级别上，C++采用和其他现代语言同样的规则：所有的二元运算都是在类型完全相同的操作数上进行运算的。只能在源代码级上混合使用不同的类型。当对表达式求值时，为了使运算确实是在类型完全相同的操作数上进行，一种数值类型的值可以（经常是这样）转换为另一种数值类型的值。

这是为了方便程序员而设计的。因此，允许编写使用混合类型的表达式而不会有语法错误。但为此要付出代价：要学习类型之间的转换规则，并且要考虑转换后的结果是否是正确的。

在使用混合类型的表达式中有三种类型转换：

- 整数升级（integral promotion）。
- 隐含类型转换。
- 显式类型转换（类型转换）。

整数升级（加宽）适用于把“小的”整数类型转换为“一般的”整数类型。这些升级适用于bool、signed字符以及short int类型的数值。从内存中被检索出来之后，这些数值总是会被升级为int以便用于表达式中。这些转换一定可以保存被升级的值，因为int类型的大小足够表示这些“更小”类型的值。在以下的例子中，将两个short类型的值相加并打印

出加法的结果及其所占空间的大小：

```
short int x = 1, y = 3;
cout <<"The sum is " << x + y <<" its size is " <<sizeof(x-y)<<endl;
                                                    // it prints 4 and 4
```

以上的计算不是在short值上进行的，而是在转换后的整数数值上进行的。其转换相当简单：在16位计算机上，简单性的原因是short和int类型所占的空间相同。在32位计算机上，要在short值上增加另外两个字节，而且用原来符号位上的值（正数为0，负数为1）来填充这两个字节。这些升级避免了值上的提升。

类似地，unsigned char和unsigned short int也可以升级为int。在32位机上，这没有任何问题，因为在这些机器上整数的范围比short的范围更大。即使是无符号的整数在16位机上，情况会有所不同。在这些机器上unsigned short的最大值是65 535，这比整数的最大值（32 767）还大。但是仍然不必担心，如果该数值不在整数范围内，编译程序会将它扩展为unsigned int。因此对程序员来说，升级是透明的。

浮点数的升级和整数升级相似。要把float数值升级为double。没有什么计算会在float上进行。当一个float值从内存里检索出来后，它会被升级为double。

整数和浮点数的升级是枯燥的、专业的和令人厌烦的。程序员应该了解它们，因为它们所耗费的时间可能对有时间紧迫性的应用有影响。例如，当一个通信应用中要处理大量的字符时，程序员可能选择把在内存中的字符当做整数，以避免每次从内存中取出一个字符数值时，都要隐式地对它们升级。这是在程序设计中常见的在时间和空间之间进行折中的典型例子。然而，有一个好处是：从程序的正确性的角度来说，整数的升级并不会对结果有什么影响，而其他的转换却会有影响。

隐含的类型转换是在出现下述情况时由编译程序完成的：

- 具有混合类型操作数的表达式。
- 赋值（根据目标的类型）。

当一个表达式含有不同大小的数值类型操作数时，会对“较小的”操作数进行加宽转换——把其数值转换为“较大的”类型的数值，然后，运算就会在两个具有相同类型的并且类型“较大的”操作数上进行。如果表达式中的运算符不止一个，表达式会根据运算符的结合性质（通常是从左向右）进行求值，而且每一步都会进行适当的转换。下面是在表达式中进行类型转换的层次结构：

```
int --> unsigned int --> long --> unsigned long --> float --> double --> long double
```

和升级类似，这些隐含的类型转换可以保存被升级的操作数。然而，要确保进行必要的转换是程序员的责任。否则可能会导致精度的丢失（见程序3-6和程序3-7）。

赋值转换把其右边的数值类型转换为左边赋值目标的数据类型。同样，运算本身（赋值）总是在类型完全相同的操作数上完成的。如果发生了截取，其精度可能会丢失，但这不是一个语法错误。许多编译程序会发出一个关于精度可能会丢失的警告，但在C++中该操作是合法的。如果这正是程序员想要的，那么就这么做。用另一句话来说，C++程序员有自己选择的权力。

除了精度的丢失以外，隐含的类型转换还与两个因素有关：运行的速度和结果的正确性。

考虑一下程序3-6中的代码，它把华氏温度转换为摄氏温度。这个程序的输出样本如图3-5

所示。

程序3-6　隐含类型转换例子

```
#include <iostream>
using namespace std;
int main()
 {
   float fahr, celsius;
   cout << "Please enter the value is Celsius: ";
   cin >> celsius;
   fahr = 1.8 * celsius + 32;                    // conversions ?
   cout << "Value in Fahrenheit is " << fahr << endl;
   return 0;
   }
```

```
Please enter the value is Celsius: 20
Value in Fahrenheit is 68
```

图3-5　程序3-6的代码，带有隐含类型转换为double，产生正确结果

字面值1.8的类型是double。类型为float的变量celsius在相乘之前被转换为double；由于数值32的类型是int，为了使加法运算在类型相同的操作数上执行，32在相加前要转换为double类型。计算的结果是double类型。由于变量fahr是float类型，因此在赋值运算之前，计算的结果又再被转换为float类型。当然，三次转换不算太多。但如果这些计算要重复很多次，就会有损于程序的性能。一个C++程序员应该留意程序的性能或者至少对讨论有关性能的问题有所准备。

这种问题的改进可以通过使用显式类型后缀，或直接在double类型中进行计算来实现。

下面是使用显式类型后缀的例子：

```
float fahr, celsius; . . .
fahr = 1.8f * celsius + 32f;        // floats are promoted to double
```

下面是使用double类型来计算的例子：

```
double fahr, celsius; . . .
fahr = 1.8 * celsius + 32.0;        // no conversions
```

即使不考虑程序的性能（我们经常忽视它），而只是希望设计出可读性强的代码，也应该记住与隐含类型转换有关的问题。例如，从摄氏到华氏温度的标准转换方法是使用系数9/5来实现的。为了举例方便，把9/5写为1.8。通常，不会考虑使用手工计算的方法来冒出错的危险，而是会像程序3-7那样由程序来实现。毕竟，在一个交互式的程序中，其执行时间主要是等待用户输入数据或为用户输出数据的时间，面少量的额外类型转换不会占用很多的时间。该程序的输出样本如图3-6所示。

程序3-7　在整数计算中丢失了精确度的例子

```
#include <iostream>
using namespace std;
int main()
 {
   double fahr, celsius;
   cout << "Please enter the value is Celsius: ";
```

```
cin >> celsius;
fahr = 9 / 5 * celsius + 32;          // accuracy ?
cout << "Value in Fahrenheit is " << fahr << endl;
return 0;
}
```

```
Please enter the value is Celsius: 20
Value in Fahrenheit is 52
```

图3-6 在延迟转换为double之后，程序3-7的代码产生了不正确的后果

错误输出的原因是没有从整数转换为double，尽管我们希望这样。由于二元运算符从左向右结合，因此是整数9被整数5除，于是结果是1。如果将这行代码写为以下形式，其结果将会有所不同：

```
fahr = celsius * 9 / 5 + 32;          // accuracy ?
```

在这里，变量celsius属于double类型。所有的计算都是在double类型上进行的。

可见，程序员需要一个能够保证所需转换发生的工具。C++为程序员提供了类型转换，它是一种显式地控制数值类型之间进行转换的方式。类型转换是一个有着高优先级的一元运算符。它带有置于圆括号内的类型名，要将它放在要进行类型转换的值的前面。例如，(double)9把整数9转换为double类型9.0；类似地，(int)1.8把double类型的1.8转换为整数1。这就是程序员描述类型转换的方法（把9转换为double类型，等等）。实际上，9并没有被改变；也就是说，它仍是一个整数。只是产生了一个double类型的新值，它在数值上等于整数9。

注意 类型转换对值进行了转换。实际上，类型转换产生了一个属于目标类型的新值，并且使用类型转换的操作数的数值对新值进行了初始化。

程序3-7中有错的那行可以用显式的类型转换改写为：

```
fahr = (double)9 / (double)5 * celsius + (double)32;
```

实际上，为了避免信息丢失问题，这样写已足够了：

```
fahr = (double)9 / 5 * celsius + 32;
```

这会把整数9转换为double类型9.0，因此整数5将会隐含地转换为double类型的5.0。

这种形式的类型转换是C++从C那里继承而来的。C++还支持另一种形式的类型转换，它类似于函数调用的语法：使用不带圆括号的类型名，但操作数要用在圆括号里。使用这种形式的C++类型转换，程序3-7的计算则变为：

```
fahr = double(9) / 5 * celsius + 32;
```

另外，C++还支持4种类型转换：dynamic_cast、static_cast、reinterpret_cast以及const_cast。这些类型转换将在以后讨论。一些程序员使用可用于表达式的显式转换（类型转换），以便向维护人员表明他们在设计时的意图。另一些程序员则觉得类型转换会影响源代码并使得维护人员的工作更困难，还有一些程序员不使用类型转换的原因是他们不愿意做额外的输入工作。

再谈一下表达式的求值。在几种不同的场合，都曾提过运算符按从左向右的次序执行，这会给人一种印象：表达式一定是从左向右求值的。但这是不对的。C++没有任何关于表达式成分求值次序的规定，只是规定了在表达式中运算符的执行顺序。

这是一个程序员经常会忽视的问题。例如，在把摄氏转换为华氏温度的表达式中，运算符从左向右求值，而数值9、5、celsius和32的求值次序如何并没有什么关系。这些计算是各自独立的。但当在其他表达式中使用有副作用的运算符时，就有关系了。比如，以下这段代码的结果会是什么呢？

```
int num = 5, total;
total = num + num++;                          // 10 or 11?
cout << "The sum is " << total << endl;
```

由于使用了一个后缀运算符，num的值在被加1前用于表达式中，因此total的值等于10。但这是在假定了表达式按照从左向右次序求值下的结果。如果它们从右向左求值，那么首先应对num++求值，值5为了用于计算而被保存起来，而num的值变为6；然后对左操作数num求值，而其值已是6，因此total的值变为11，而不是10。

在有些计算机上，上例所得的结果是10，但在另外一些计算机上运行，结果可能也是10，但这并不意味着什么，C++并没有保证任何表达式的成分都按从左向右的次序求值，那么，解决的方法是什么呢？答案是不要在表达式中使用有副作用的东西。如果我们希望在所有的计算机上结果都是10的话，就应像下面那样书写：

```
int num = 5, total;
total = num + num;                        // 10, not 11 num++;
cout << "The sum is " << total << endl;
```

如果希望在所有的计算机上运行结果都是11的话，也不难，可写为：

```
int num = 5, total;
int old_num = num;  num++;
total = num + old_num;                    // 11, not 10
cout << "The sum is " << total << endl;
```

因此，总是有办法显式地表达用户的意图。

3.6 小结

关于C++的类型和表达式的求值问题就讨论到此，可见，认真考虑所使用类型的取值范围总是一个好习惯。无论从可移植性还是从结果正确性的观点来看，这都是很重要的。除非有特别的原因要使用其他的类型（例如，老板吩咐要这样），否则，最好使用int类型和double类型；但要保证它们能够正确地工作。

这一章讨论了很多基本的内容，要熟悉它们可能要花一些时间。通过例子进行实践，并把第2章中所学的内容作为基础。不要太多太快地使用那些高级的语言特征。

如果能够适应的话，可以自由地在表达式中混合使用各种数值类型；但要考虑其类型转换的情况以及对性能和正确性的影响。适当地使用显式类型转换，不要把有副作用的表达式作为其他表达式的一部分。要避免不必要的复杂性：它会使编译程序、维护人员以及程序员自己都感到困惑。

要确切地知道自己正在做些什么。

第4章 C++控制流

在第3章中，我们已经讨论了C++程序设计的基本内容，包括：数据类型以及把各种数值组合成表达式和语句的运算符。本章，我们将讨论程序设计的深层次的内容：把语句组合成为能够根据外部环境做出判断并执行不同代码段的算法。

正确使用控制结构是决定代码质量的重要因素之一。当执行的流程是顺序时，语句以固定的顺序被逐个地执行，维护人员理解程序代码就会相对容易。对于每一段代码来说，由于只存在一组初始条件，因此只有一个计算结果。但顺序的程序太简单了，它们不能实现许多功能。每一个实际的程序都要在某些条件下执行某些代码段，而在另一些条件下执行另一些代码段。控制应该能从一段代码转移到另一段代码。从控制结构的观点来看，程序设计语言越灵活，程序员手中的权力就越多。

当一段代码不是在某一段代码之后执行就是在另一段代码之后执行时，就存在多于一组的初始条件，因此计算结果可能不止一个。选择的多就意味着难度变大。程序员在编写代码时易出现错误，维护人员在阅读程序和修改程序时也会发生错误。因此现代程序设计语言试图在控制流从一段代码转移到另外一段代码时，限制程序员所做的工作，这个方法叫做结构化程序设计。程序员只能使用一些符合结构化要求的控制结构（循环和条件语句），使得每一段代码只有1个（或两个）入口和1个（或两个）出口。

C++采用了以下的折中方法：它提供了一组丰富的、可以在程序中改变控制流的控制结构这些结构灵活且强有力，能够支持程序中所要做出的复杂判断。同时，它对防止出现难以理解和维护的复杂设计也提供了足够的限制。

4.1 语句和表达式

不像其他语言那样，C++中表达式和可执行语句之间的区别很小：任何表达式都可以通过在其后添加一个分号而转换为语句。下面是表达式和可执行语句的例子。

```
x * y           // valid expression that can be used in other expressions
x * y;          // valid statement in C++, but quite  useless
a = x * y       // valid expression that can be used in others (do it with caution)
a = x * y;      // valid C++ statement, useful and common
x++             // valid expression that can be used in others (do it with caution)
x++;            // valid C++ statement, common and useful
foo()           // call to a function returning a value (a valid expression)
foo();          // call to a function with return value unused (a valid statement)
;               // null statement, valid but confusing
```

和其他语言一样，C++的语句是按照它们在源代码中出现的次序顺序地执行的。从逻辑上说，每一条语句都是一个不可以中断执行的完整单位。

可执行语句可以组成一个块（复合语句）。块应该用一对花括号来定界。从语法上说，一个块语句被当做一条单独的语句，而且可以用于任何一条单独语句允许出现的地方。在块中的每一条语句都必须以分号结束，但是在块的闭括号后不应该跟着一个分号。

将语句合并成一个块有两个重要的优点。首先，在语法上只允许出现一条语句的地方可以使用含有几条语句的块。其次，可以在块中定义局部变量。这些变量的名字不会和其他地方定义的变量名字发生冲突。第一个特性对于编写控制语句来说是很重要的，没有它就不能写出任何理想的程序。第二个特性对编写函数来说很重要，没有它，同样不能写出任何理想的程序。下面是一条复合语句的一般形式：

```
{ local definitions and declarations (if any);
  statements terminated by semicolons; }          // no ; at the end
```

复合语句可以用作函数体、函数体内的嵌套块和控制语句体。如果我们在复合语句后面加了分号，在大多数情况下不会有问题，但会得到一条不产生任何目标代码的无用的空语句。然而有时这样会改变代码的含义。最好不要在闭括号后使用分号（要记住的是要求有分号的例外情形，比如在类定义和其他的一些例子中）。

复合语句是在上一条语句之后和下一条语句之前被作为一条单独语句来求值的。在块内，一般的控制流也是按照语句的字面出现次序顺序执行的。

C++提供了一组标准的控制语句，它们可以在程序中改变顺序执行的流程。这些控制语句包括：

- 条件语句：if、if-else语句。
- 循环语句：while、do-while、for语句。
- 转移语句：goto、break、continue、return语句。
- 多入口代码：有case分支的switch语句。

在条件结构中，所控制的语句是被求值一次还是被跳过，要取决于该条件的布尔表达式。在循环中，它的语句是被求值一次、多次或是跳过，要取决于循环的布尔表达式。布尔表达式是一个返回值为true或false的表达式，经常也称为逻辑表达式、条件表达式或表达式。在C++中，任何表达式都可以作为布尔表达式。这扩展了C++中条件语句和循环语句的灵活性。在其他语言中，在要求使用布尔表达式的地方使用非布尔表达式会产生语法错误。

在一个开关分支语句（switch）中，根据一个整数表达式的求值结果选择一个适当的case分支（从几个分支中）来执行。转移语句无条件地改变控制流。它们通常与其他某个控制结构（条件、循环、开关分支等语句）结合在一起使用。

总之，对于所有的控制结构来说，其作用域仅是单个语句。当算法的逻辑要求将几条语句的执行作为逻辑表达式的测试结果时，可以使用置于花括号内的复合语句。在闭括号后不应该有分号，但块中的每一条语句（包括最后一个）应该跟着一个分号。

在本章的后续部分，我们将详细地讨论每一种C++的流控制语句；给出在使用这些控制语句编写C++代码时，要做什么和不要做什么的示例以及具体的建议。

4.2　条件语句

条件语句可能是C++程序中最无所不在的控制结构。编写代码时通常都会有一些使用条件语句才能完成的工作。

编写C++代码时有几种形式的条件语句可供选择。条件语句的复杂形式需要不断地测试，但使用它们可使源代码更加精简和美观。

4.2.1 条件语句的标准形式

C++条件语句的最一般形式有两个分支：true分支和false分支。当执行条件语句时，只执行其中一个分支。

下面是条件语句在上下文中的一般形式：它位于两条语句之间。

```
previous_statement;
if (expression)                // no 'then' keyword is used in C++
  true_statement;              // notice the semicolon before 'else'
else
  false_statement;             // notice optional indentation
next_statement;
```

关键字if和else必须用小写。在C++中没有关键字‘ther’，表达式必须置于圆括号中。

在previous_statement（它可以是任何的语句，包括控制结构之一）执行之后，就对圆括号内的表达式求值。逻辑上，它是一个布尔表达式；要计算条件的值是true还是false。当条件表达式为true时，就执行true_statement并跳过false_statement。当条件为false时，就执行false_statement而跳过true_statement。由于我们是在用C++语言而不是Pascal、Basic、Java或PL/I语言，因此条件表达式不一定是布尔类型。它可以是任意复杂度的任何表达式。在对它求值时，把所有的非0值（甚至不要求是整数）作为true，而把0值作为false。

在执行了两条语句（true_statement或false_statement）中的某一条之后，就无条件地执行next_statement。同样，next_statement可以是任何语句，包括控制结构。

程序4-1给出了一个例子，它提示用户输入摄氏温度，然后接收输入数据，最后告诉用户该温度是否有效（高于绝对温度0）。在摄氏温度中，绝对0度是零下273度，或说-273℃。

程序4-1 一个条件语句

```
#include <iostream>
using namespace std;

int main ()
{
  int cels;
  cout << "\nEnter temperature in Celsius: ";
  cin >> cels;
  cout << "\nYou entered the value " << cels << endl;
  if (cels < -273)
    cout <<"\nThe value " <<cels <<" is invalid\n" // no ;
         <<"It is below absolute zero\n";          // one statement
  else
    cout <<cels<<" is a valid temperature\n";
  cout << "Thank you for using this program" <<endl;
  return 0;
}
```

请注意，这里用cout对象打印位于双引号里的字符串的开头和结尾的两个换行转义字符。

也请注意在程序结尾使用了endl操纵符。如果所用的操作系统没有使用缓冲区输出，换行转义字符‘\n’和endl操纵符之间就没有什么不同。若用到缓冲区，endl把输出送给缓冲区并“冲洗”缓冲区；也就是说，从缓冲区完成实际的输出。然而，‘\n’只是将输出送给缓冲区，只有当缓冲区满时才“冲洗”缓冲区。这种做法有时可以改进程序的性能，但很多程序员并不关心这一差别。

该程序的输出如图4-1所示。

```
Enter temperature in Celsius: 20

You entered the value 20
20 is a valid temperature
Thank you for using this program
```

图4-1　程序4-1中程序的输出结果

请注意条件语句的一般形式以及程序4-1中所使用的缩进。通常把关键字if和else排版为与条件语句的前后两个语句对齐。通常把true_statement和false_statement向右缩进若干个空格。对于维护人员（以及调试时的代码设计人员）来说，这会使控制流更加清晰。缩进多少是个人的品味，本书采用缩进两个空格。如果缩进更多，就会缩短了该行；特别是在使用嵌套控制结构时，此时true_statement或false_statement（或两者）本身也是条件语句、循环语句或开关选择语句。

注意，当输入温度无效时，程序输出两行信息，这时相应的代码应为：

```
cout <<"\nThe value " <<cels <<"is invalid\n";   // ; at end
cout <<"It is below absolute zero\n";            // two statements
```

用这种方式编写时，这两条语句必须置于复合语句的花括号中。原因是：当代码是条件语句的一部分时，条件语句的每个分支只能容纳一条语句，不能容纳两条语句。

程序4-1使用了一种不同的技术，语句cout可以取任意长度，并且可跨越源代码的任意多行，只要该行是在语句的中间断开，而不是在串的中间断开。这意味着将一行在字符串中间断开是不正确的。然而，允许有必要时将字符串一分为二。

条件语句的false_statement是可选的。如果只有在布尔表达式的值为true时才执行某个动作，那么false_statement就可以省略。下面是没有false_statement的条件语句的一般形式：

```
previous_statement;
if (expression)
  true_statement;
next_statement;
```

这个条件语句既没有then也没有else关键字。程序4-2给出的是程序4-1的一个“简化”版本。当输入数据无效时（也就是温度低于绝对0度时），它会向用户发出警告信息，但程序会继续执行其工作（为了简单起见，在这里省略了有关的工作，并且只给出了结论）。其运行结果如图4-2所示。

程序4-2 没有else部分的一个条件语句

```
#include <iostream>
using namespace std;
#define ABSOLUTE_ZERO -273

int main ()
{
 int cels;
 cout << "\nEnter temperature in Celsius: ";
 cin >> cels;
 cout << "\nYou entered the value " << cels << endl;
 if (cels < ABSOLUTE_ZERO)
  cout <<"\nThe value " <<cels <<" is invalid\n"
       <<"It is below absolute zero\n";          // one statement
 cout << "Thank you for using this program" <<endl;
 return 0;
}
```

```
Enter temperature in Celsius: 20

You entered the value 20
Thank you for using this program
```

图4-2 程序4-2的输出结果

像前一个程序一样，关键字if与前后两个语句对齐；true子句的代码向右缩进，以表明控制结构。

注意对绝对0使用的是符号常量而不是程序4-1中的字面值。对每一个字面值都使用符号常量并在程序中把它们的定义放在一起，这是程序设计中的一种好的做法。它使维护工作变得容易：维护人员知道在哪里可以找到这些值，并且改动一次就会对在程序中该字面值的每一次出现都有效。这比在代码中逐个搜寻该字面值的出现，并可能因为遗漏修改而引起错误要好得多。在这个小型的程序中，-273是程序中使用的惟一数值，它只使用了一次。如果想改动它，那么在程序中的任何一个地方改动它都是一样的。毕竟，程序维护时，修改绝对0的数值会有多少次呢？因此，使用符号常量还是使用字面值，在这里都是一样的。然而，使用符号常量是一种更好的做法。

注意 如果有必要的话，条件语句中的true_statement和false_statement可以是复合语句。

程序4-3给出的是对程序4-1修改后所得的程序。在true分支中使用了两条语句，在false分支中也使用了两条语句。请注意关键字const的使用，以前曾提过，在C++中这是一个比使用#define预处理程序指令更常见的技术。程序的输出如图4-3所示。

程序4-3 分支语句为复合语句的条件语句

```
#include <iostream>
using namespace std;
const int ABSOLUTE_ZERO = -273;

int main ()
{
```

```
    int cels;
    cout << "\nEnter temperature in Celsius: ";
    cin >> cels;
    cout << "\nYou entered the value " << cels << endl;
    if (cels < ABSOLUTE_ZERO)
     { cout <<"\nThe value " <<cels <<" is invalid\n";
       cout <<"It is below absolute zero\n"; }              // a block
    else
     { cout <<cels<<" is a valid temperature\n";           // a block
       cout << "You can proceed with calculations\n"; }
    cout << "Thank you for using this program" <<endl;
    return 0;
}
```

```
Enter temperature in Celsius: 20

You entered the value 20
20 is a valid temperature
You can proceed with calculations
Thank you for using this program
```

图4-3　程序4-3中程序的输出结果

复合语句必须使用开括号和闭括号。在复合语句中的每一个语句都再一次向右缩进以表明它们是顺序执行的。这可以帮助维护人员理解设计人员在实现时的意图。有些程序员让每一个复合语句的开括号和闭括号独占一行。他们觉得这对强调代码的结构有帮助。这样做是否值得还不确切，因为会导致程序在垂直方向上变长，于是要把握代码的主要含义会更难（特别是在要根据屏幕显示而不是打印出来的文本来调试程序时）。因此本书很少使用垂直方向的空白间隔。

4.2.2　条件语句中的常见错误

条件语句增加了代码的复杂度。条件语句中的错误通常难以发现。如果运气好的话，这些错误只是一些语法错误。通常，这些错误都会导致程序执行不正确。由于不是每一次都执行条件语句中的所有语句，因此需要通过额外的规划和运行测试来发现这些错误。

当程序设计人员把意图和知识传达给维护人员时，经常会发生错误，这体现在错误的缩进排版或者复合语句定界花括号的错误使用。

遗漏花括号是控制结构中常见的错误。假如把程序4-3中的条件语句写成这样：

```
if (cels < ABSOLUTE_ZERO)
  { cout <<"\nThe value " <<cels <<" is invalid\n";
    cout <<"It is below absolute zero\n"; }          // a block
else
    cout <<cels<<" is a valid temperature\n";         // no braces
    cout << "You can proceed with calculations\n";
```

这个程序初看起来没问题，编译、运行也没问题。至少当输入数据是20时，程序的输出和图4-3是完全相同的。然而，如果输入-300时，其输出将是图4-4那样。

请注意输出是不正确的。原因是，缩进只对阅读者可见，而对编译程序是不可见的。尽管缩进表示了两个cout语句都属于else分支，但编译程序却不这样看。由于没有花括号，

编译程序会认为第二个cout语句就是下一条语句而不是false分支语句的一部分。编译程序会理解为：

```
Enter temperature in Celsius: -300

You entered the value -300

The value -300 is invalid
It is below absolute zero
You can proceed with calculations
Thank you for using this program
```

图4-4 修改了程序4-3的程序后的输出结果

```
if (cels < ABSOLUTE_ZERO)
  { cout <<"\nThe value " <<cels <<" is invalid\n";
    cout <<"It is below absolute zero\n"; }      // a block
else
    cout <<cels<<" is a valid temperature\n";     // no braces
cout << "You can proceed with calculations\n";   // next_statement
```

幸亏if语句的true分支中一个类似的错误将产生以下的语法错误：

```
if (cels < ABSOLUTE_ZERO)
  cout <<"\nThe value " <<cels <<" is invalid\n";
  cout <<"It is below absolute zero\n";           // this is nonsense
else
  { cout <<cels<<" is a valid temperature\n";     // a block
    cout << "You can proceed with calculations\n"; }
```

这时，编译程序会警告关键字else的位置有误，因为编译程序会将代码理解成：

```
if (cels < ABSOLUTE_ZERO)                         // an 'if' without an 'else' is ok
  cout <<"\nThe value " <<cels <<" is invalid\n";
cout <<"It is below absolute zero\n";             // this is nonsense
else                                              // this 'else' does not have the 'if'
 { cout <<cels<<" is a valid temperature\n";       // a block
   cout << "You can proceed with calculations\n"; }
```

编译程序会认为第一条cout语句属于没有else子句的if语句，这是完全合法的。编译程序会认为第二条cout语句是下一条语句，这也是允许的。当编译程序发现关键字else时，会认为它是多余的。

注意 要确保正确地使用花括号。它们是一个很常见的错误根源。

一个相关的问题是C++语句结尾处分号的使用。以前曾提过，遗漏C++语句后的分号会导致麻烦。C++初学者通常会努力记住这个规则。有些程序员太过注意这个问题而导致他们在程序每一行的结束处都加上一个分号，而不管是否有必要。如果在源代码中使用了多余的分号，就会得到一个什么也没做的空语句，尽管这在大多数情况下是无害的。（这个结论是观察所得的经验，而不是从某本指南书上引用而来的。）

然而，多余的分号并不总是无害的。假设把程序4-2中的#define指令写成这样：

```
#define ABSOLUTE_ZERO -273;       // incorrect #define
```

这当然是错误的。这里不应该有分号（但在程序4-3中的常量定义末尾应该有一个分号）。

然而，编译程序不会指出这一行有错，而是指出条件语句写错了。请注意#define指令的作用是进行字面替换。每当预处理程序在程序中发现标识符ABSOLUTE_ZERO时，就会把其值替换到源代码中去。现在的值是-273；而不是-273。对于预处理程序来说，这是完全合法的；但编译程序从预处理程序那里得到了要处理的以下条件语句：

```
if (cels < -273;)                          // semicolon in expression: error
 cout <<"\nThe value " <<cels <<"is invalid\n"
      <<"It is below absolute zero\n";   // one statement
```

表达式后面的分号把表达式转变为一条语句。编译程序就会指出在表达式cels<ABSOLUTE_ZERO中含有一个多余的分号。可以看到程序4-2中的这个表达式并没有分号。于是我们可能会以为错误是由其他地方引起的，并且开始在这一行的附近对一切有怀疑的东西进行修改。怀疑越多，情况就越糟。由于出错的地方（#define指令）与显示有错的地方（条件语句）之间距离太大，使得难以对情况进行分析。因此使用关键字const比使用#define指令会更好一些。

有时可能会在条件表达式所在行的结束位置加上一个分号。假如把程序4-3中的条件语句写成这样：

```
if (cels < ABSOLUTE_ZERO);                              // the true branch
  { cout <<"\nThe value " <<cels <<" is invalid\n";   // next_statement
    cout <<"It is below absolute zero\n"; }           // a block
else                                                    // nonsense for the compiler
  { cout <<cels<<" is a valid temperature\n";
    cout << "You can proceed with calculations\n"; }
```

这是一个语法错误。由于它会经常出现，编译程序不能够正确地指出出错的位置，只会指出关键字else的位置有误。对编译程序来说，条件表达式之后那个多余的分号使该行成为一个完整的语句，它没有做什么操作，但这并不是C++的问题。下面是编译程序对以上代码的理解：

```
if (cels < ABSOLUTE_ZERO)
  ;                                                     // it does not do much
{ cout <<"\nThe value " <<cels <<'is invalid\n";
  cout <<"It is below absolute zero\n"; }               // next statements
else                                                    // misplaced 'else'
 { cout <<cels<<" is a valid temperature\n";
   cout << "You can proceed with calculations\n"; }
```

编译程序会认为条件语句没有else分支，并且在条件语句之后是含有两条语句的块，接着是关键字else，于是认为此行有错，却没有指出真正出错的那行。如果出现这样的错误，请不要花费太多的时间去理解编译程序给出的出错信息或重新安排代码的结构。

假设在程序4-2的程序中，在逻辑表达式之后加上一个分号，程序4-2的条件语句将变为：

```
if (cels < ABSOLUTE_ZERO);                  // this is definitely harmful
 cout << "\nThe value " <<cels <<" is invalid\n"
      << "It is below absolute zero\n";
```

这个条件语句没有else子句。这里并没有将else写错位置，而且这一程序编译时没有出现问题，编译程序不会发出错误的警告。在调试和测试中，就要十分小心。如果将值20作为输入数据运行该代码的话，其结果会与图4-2不同。修改之后的程序输出如图4-5所示。

这种错误在调试时会难以发现。当程序产生多个正确的输出时，这个细微的错误不会引

起程序员的注意。我们应该像对待花括号那样严格地对待分号。

```
Enter temperature in Celsius: 20

You entered the value 20

The value 20 is invalid
It is below absolute zero
Thank you for using this program
```

图4-5 修改程序4-2中程序后的输出结果

使用控制结构会引起一个新的问题，就是程序的测试问题。实际上，这不是一个新问题；但对于控制语句，要求程序员要做好规划、要有技巧并且要提高警惕。当测试顺序结构的程序时，通常运行一次程序就够了。如果程序不正确，结果就会有错。这样的程序运行一次就足够了，除非程序员在打瞌睡或在思考其他事件，或者忙于从一件事情转向另一些事情。

注意 前一章中所有的程序都是执行路径惟一的顺序结构程序，因此可通过运行一次来判断程序是否正确。

即使对于顺序结构的程序，有时只运行一次也不足以表明程序的正确性。至少应该用两组数据去运行它们，这一点很重要。原因是存在偶然正确的可能性。为了说明这一点，考察一下程序3-7中将摄氏温度转换华氏温度的例子。你应该还记得，以下的计算语句是不正确的：

```
fahr = 9 / 5 * celsius + 32;          // accuracy ?
```

当程序员设计要输入的测试数据时，最重要的是考虑简化人工计算。这是十分合理的，因为算法通常很复杂，会使得一般的人工计算都难以正确地完成。在这种情况下，理想的方法是让程序员将0作为输入数据来测试程序3-7的程序。其结果显示如图4-6所示。可以看到，它们是正确的。因此即使对于顺序结构的代码段，一组测试数据也往往是不足够的。

```
Please enter the value is Celsius: 0
Value in Fahrenheit is 32
```

图4-6 程序3-7中程序的输出结果

我们再回到程序4-1中的程序。用输入数据20（和在图4-1里的一样）来运行程序是否就足够了呢？很明显是不够的，因为在程序中还有一些语句在程序运行过程中从未执行过。如果这些语句要把钱转到程序员的账户，如果要发射一枚洲际导弹，结果会怎样呢？程序会崩溃吗？还是会悄悄地产生不正确的输出呢？测试的第一条原则是该组测试数据应该使程序的每一条语句至少执行一次（或者更多次，如果想防止隐藏在偶然正确结果里的错误并保护程序的话）。因此，除了图4-1中的那个数据外，程序4-1至少需要做第二次测试。

图4-7给出了程序4-1程序的第二次测试的结果。可以看到其结果是正确的，这就增强了我们对该程序正确性的信心。其输出结果证实了条件语句的两个分支的确发生了作用，并且它们能正确工作。

```
Enter temperature in Celsius: -300

You entered the value -300

The value -300 is invalid
It is below absolute zero
Thank you for using this program
```

图4-7　程序4-1程序的第二次运行

这个测试充分吗？可能不。如果绝对零度的数值输入错了，比如是-263而不是-273，两次测试的结果还会是正确的。因此我们需要的第二条测试原则是：测试数据应该能够测试条件表达式的边界。也就是说要用-273作为输入数据。如果绝对零度被输入为-263，程序就会打印出温度-273是不正确的这一错误的输出。因此用-273作为输入数据就可以发现在图4-2中输入数值为0时所不能发现的错误。

但这还没有结束。如果将绝对零度写成-283而不是-273，情况又会怎么样呢？用-273作为输入数据将不会发现这个错误，因为条件-273<-283的值为false，于是程序将（正确地）打印出这是一个正确的温度。于是我们需要的第三条测试原则是：必须测试条件表达式的边界为true以及为false这两种情况。

对于整数的情形，就意味着要使用-274作为输入。对于浮点数的情形，程序员就要选择稍大于边界的值，如-273.001，或者一个有实际应用意义的其他值。

通常，如果代码含有条件x<y，它必须用两种情况来测试：一种情况是x等于y（其结果应该为false）；另一种情况是，如果是整数则令x等于y-1，如果是浮点数则令x等于y减去一个小的值（其结果应该是true）。

类似地，如果代码含有条件x>y，它也必须用两种情况来测试：一种是x等于y（其结果应该为false）；对于整数来说，另一种情况是x等于y+1，或者对于浮点数来说，是x等于y加上一个小的数值（其结果应该为true）。

不幸的是，以上还不是全部的测试情形。这些原则对于含有相等的条件来说不起作用。如果代码含有条件x<=y，测试x是否等于y应该返回true，而不像x<y中那样返回false。为了测试结果为false的情形，代码应该用x等于y+1（或y加上一个小的数）来测试。类似地，如果代码含有条件x>=y，测试x是否等于y应该返回true而不是false，其第二个测试情形应该是x等于y-1（或y减去一个小的数）。

这使得测试变得相当复杂。程序的每一个条件都必须单独地测试，并且每一个条件要测试两种情形，于是测试情形的数目会很大。某些程序员没有耐心去分析、设计、运行和检查为数众多的测试情形。他们只是有限地进行可视的代码检查，这是令人遗憾的。代码检查虽然有用，但却不是查错的可靠工具。

当要测试数值是否满足相等（或不等）关系时，测试情形会变得更加复杂。程序4-4提示用户输入非0整数，程序接受输入值，然后检查该数是否为0（防止被0去除）。如果该数不是0，用户会由于正确地输入而被肯定，然后程序会计算输入值的倒数以及平方。如果输入值是0，程序就会认为用户不遵从指令。当然，这只是一个简单的例子，但在情况不太复杂的情况下说明了有关的问题。

程序4-4 检查值是否相等（不正确的版本）

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout << "\nPlease enter a non-zero integer: ";
  cin >> num;
  if (num > 0)                                  // it should be (num != 0)
   { cout <<"\nYou followed the instructions correctly";
     cout <<"\nThe inverse of this value is " << 1.0/num;
     cout <<"\nThe square of this value is " << num * num; }
  else
     cout <<"\nYou did not follow the instructions";
  cout << "\nThank you for using this program" <<endl;
  return 0;
}
```

请注意，如果用1/num代替1.0/num，其输出将是不正确的，因为整数除法会对结果进行截取。图4-8给出了输入值为20的程序测试结果，它显示了正确的输出。

```
Please enter a non-zero integer: 20

You followed the instructions correctly
The inverse of this value is 0.05
The square of this value is 400
Thank you for using this program
```

图4-8 程序4-4的第一次测试的输出结果

一种测试情形显然不够，因此还要用违反程序指令的输入数据来测试程序，以保证if语句的else分支能被执行。从图4-9中可以看到，程序也通过了这次的测试，但它指出用户没有遵从指令。

```
Please enter a non-zero integer: 0

You did not follow the instructions
Thank you for using this program
```

图4-9 程序4-4的第二次测试的输出结果

但是请稍候，这个程序是不正确的！输入该程序时犯了一个错误：将if的条件输入为num>0而不是num!=0。顺便说一下，将关系运算符输错是很常见的。在编写算术应用程序时，程序员有时会忘记正确地实现和测试与负数相应的行为。为了说明这种错误，下面输入一个负数来进行相应的第三次测试，结果如图4-10所示。

可以看到该程序会警告用户输入有错，而不是接受输入。在程序4-5中也可以看到这个程

序的正确版本。

```
Please enter a non-zero integer: -20

You did not follow the instructions
Thank you for using this program
```

图4-10　程序4-4的第三次测试的输出结果

程序4-5　检查数值是否不相等（正确版本）

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout << "\nPlease enter a non-zero integer: ";
  cin >> num;
  if (num != 0)                                    // now this is correct
   { cout <<"\nYou followed the instructions correctly";
     cout <<"\nThe inverse of this value is " << 1.0/num;
     cout <<"\nThe square of this value is " << num * num; }
  else
     cout <<"\nYou did not follow the instructions";
  cout << "\nThank you for using this program" <<endl;
  return 0;
}
```

于是我们得到另外一条测试原则：对于使用了相等运算符的条件表达式，必须进行三次测试：一次测试是否相等（它应该返回true），另外两次测试边界上每一边的不相等（这些测试应该返回false）。对于使用了不相等运算符的条件表达式来说，也有同样的原则：对相等的测试应该返回false，两次不相等的测试应该返回true。

提示　对于有关系运算符的if语句，使用在边界上和接近于边界的测试值；使用远离边界的测试值会遗漏错误。

以上的测试原则总结在表4-1中。其中条件表示为x　op　y，运算符op可以是‘>’、‘<’、‘>=’、‘<=’、‘==’和‘!=’。对于每一个运算符，该表格都列出了条件表达式的测试情形和相应的期望值。这里假定x和y的值都是整数。对于浮点数，应该用一个小的增量来代替1。对于非数值数据的相等或不相等的测试，测试两次就足够了，不需要测试三次。

表4-1　简单条件表达式的测试情况

表达式	测试	结果
x < y	x等于y	False
	x等于y - 1	True
x >= y	x等于y	True

（续）

表达式	测试	结果
	x等于y - 1	False
x > y	x等于y	False
	x等于y + 1	True
x <= y	x等于y	True
	x等于y + 1	False
x == y	x等于y	True
	x等于y + 1	False
	x等于y - 1	False
x != y	x等于y	False
	x等于y + 1	True
	x等于y - 1	True

可以看到x<y和x>=y的测试情形是一样的，但其结果恰好相反。x>y和x<=y的测试情形也是一样的，但其结果也互为相反。类似地，x==y和x!=y的测试情形相同而结论相反。这就意味着x<y和x>=y是互为相反的情形。每一个写成x<y的条件都可以重写成!(x>=y)，反之亦然，每一个写成x>=y的条件也可以写为!(x<y)。当前一个条件为true时，第二个条件则为false；反之亦然。

类似地，条件x>y和x<=y也是互为相反的情形。条件x==y和x!=y也是一样。当每一对中的一个条件为true时，另一个则为false。

熟练掌握对逻辑条件的否定运算是程序设计的一个重要技巧。对于条件语句而言，这是有关正确书写代码格式的问题。例如，如果true分支有很多复杂的语句而false分支只有一两条语句，则false分支很可能在代码中被遗漏，于是有些程序员就喜欢把较短的语句序列作为条件语句的true分支。例如，程序4-5中的条件语句可以写成这样：

```
if (num == 0)                          // negation of (num != 0)
  cout <<"\nYou did not follow the instructions";
else
  { cout <<"\nYou followed the instructions correctly";
    cout <<"\nThe inverse of this value is " << 1.0/num;
    cout <<"\nThe square of this value is " << num * num; }
```

以前曾提过，不要把运算符‘= =’误写为运算符‘=’，这一点很重要。这也是一个难以发现错误的常见原因。例如，容易将上面的条件语句写成：

```
if (num = 0)                           // this is perfectly valid in C++
  cout <<"\nYou did not follow the instructions";
else
  { cout <<"\nYou followed the instructions correctly";
    cout <<"\nThe inverse of this value is " << 1.0/num;
    cout <<"\nThe square of this value is " << num * num; }
```

这段代码不会产生任何语法或运行错误。某些编译程序可能会发出一个警告，但总的来说，这是合法的C++用法。某些程序员对这种可能出现的错误非常不满，以致他们会把数值写在比较式子的左边而把变量写在右边，例如：

```
if (0 == num)                  // you will not use a constant as lvalue
  cout <<"\nYou did not follow the instructions";
else
```

```
{ cout <<"\nYou followed the instructions correctly";
  cout <<"\nThe inverse of this value is " << 1.0/num;
  cout <<"\nThe square of this value is " << num * num; }
```

如果把这个比较式子误写为赋值运算0=num，编译程序就会认为它有错，因为C++的数值常量没有可供程序操作的地址（它们只能是右值），尽管它们和其他数值一样保存在内存中。程序设计语言在设计上的一个共同趋势是，尽可能在编译阶段而不是运行阶段发现更多的错误，但并不总是这样的。记得有一次作者在PDP-11上使用FORTRAN语言工作时，通过某种方式把常量1的值设置成了2。因此，每当作者使用1时，编译程序总是在使用值2。这使所有的循环变得不符合逻辑，而作者却不明白这到底是为什么。

编写逻辑条件的另一种常见的技术是利用以下这一事实：在C++中任何非0数值被看做true，而0值被看做false。例如，很多程序员会把程序4-5中的条件语句写成这样：

```
if (num)                    // a popular C++ idiom, same as if (num!=0)
  { cout <<"\nYou followed the instructions correctly";
    cout <<"\nThe inverse of this value is " << 1.0/num;
    cout <<"\nThe square of this value is " << num * num; }
else                        // the 'else' should be closer to the 'if'
    cout <<"\nYou did not follow the instructions";
```

我们应该熟悉这个C++的习惯用法，因为它非常流行。如果使用这个逻辑条件的否定形式（也就是num==0），很多程序员会把条件语句写成：

```
if (!num)                 // a popular C++ idiom, same as: if (num == 0)
  cout <<"\nYou did not follow the instructions";
else                      // the 'else' should be closer to the 'if'
{ cout <<"\nYou followed the instructions correctly";
  cout <<"\nThe inverse of this value is " << 1.0/num;
  cout <<"\nThe square of this value is " << num * num; }
```

注意，写成if!(num)...是不正确的，因为逻辑条件必须置于圆括号内。人们很容易滥用这个特征并编写出一些维护人员要花一段艰苦的时间才能理解的代码。

至今为止，我们所讨论的例子都是相当简单的。为了实现更加复杂的处理，可以使用复合的和嵌套的条件语句。在复合条件语句中，不仅要测试复合条件中为真和假的结论，还要测试每一种使结论为真和为假的原因。例如，考察以下的条件语句，其中processOrder()是一个在其他地方定义的函数。

```
if (age > 16 && age < 65)
processOrder();
else
cout << "Customer is not eligible\n";
```

我们只能用一种方式来测试这个条件的true分支：方法是把age>16和age<65都设置为true。我们可以用两种方式来测试false分支：方法是把age<65设置为false（例如age是65）或把age>16设置为false（当age是15时）。选择哪一个呢？如果只使用第一种方式，若第一个条件不正确地设置为true，将不会显示有错。例如，写为age>0并不会有错。如果只使用第二种方式，若第二个条件为true但却不正确时，也不会发现错误。例如，写为age<250并不会有错。因此这两种经过条件语句的false分支的情形都应该测试。这比单个条件语句会有更多的测试情形，但这是自然的。把单个条件设为true或false的测试情形应该根据表4-1来进行设计。

注意，我们没有对经过false分支的第三种情形进行测试，其中age>16和age<65均为false。有些程序员证明可以跳过这个组合，因为这些条件是相关的：它们的真值依赖于同一个变量age的值。由于依赖于变量age的值，这些条件可以同时为true（在取值范围的中部）或者其中有一个为false（低于或高于取值范围），但它们不可能同时为false：其值不可能同时低于和高于取值范围。然而，这并不是问题所在。对于逻辑与操作，我们已经单独测试了这些条件的false值，再测试它们的组合是浪费时间和金钱的。

类似的情形适用于对逻辑或复合条件的测试。考虑下面比较两个浮点数值的例子：

```
if (amt1 < amt2 - 0.01 || amt1 > amt2 + 0.01)      // is difference more than 1 cent?
  cout << "Different amounts\n";
else
  cout << "Same amount\n";
```

我们只能以一种方式测试这个语句的false分支：把两个条件都设置为false。我们可以以两种方式测试true分支：把第一个条件设置为true或把第二个条件设置为true。选择哪一种方式呢？答案和逻辑与运算符的情形一样：根据表4-1的原则，两个情形都要进行测试，以保证两个条件都经过了充分的测试。

这里我们不必测试经过true分支的第三种情形，因为这两个条件是有联系的（它们都依赖于变量amt1和amt2的值），它们不可能同时为true。即使这两个条件没有联系，这种测试也是多余的。

表4-2给出了在测试复合条件中必须包含的测试情形。

表4-2 复合条件的测试设计

操 作	第一个条件	第二个条件	结 果
AND	True	True	True
	True	False	False
	False	True	False
OR	True	False	True
	False	True	True
	False	False	False

前面曾经提过，复合语句中的诸条件是否相关并不会对测试的策略有多大的影响。例如，考察以下带有独立条件的条件语句。这里，函数processPreferredOrder()和processNormalOrder()在程序的其他地方定义，并被条件语句的不同分支所调用。如果以前的买卖额超过1 500美元以及当前的购买总额达200美元，则该顾客将会得到优惠的待遇。

```
if (amount > 200 && previous_total > 1500)
  processPreferredOrder();
else
  precessNormalOrder();
```

为了测试这个代码，我们必须设计三种情形。一种测试情形应该经过true分支，此时anount>200和previous_total>1500都为true（比如amount=200.01，previous_total=1500.01）。另外两种测试情况应该经过false分支，其中一种应该把条件amount>200设置为true而把previous_total>1500设置为false（比如，amount=200.01，previous_total=1500.00），另外一种则应该把amount>200设置

为false而把previous_total>1500设置为true（比如，amount=200.00，previous_total=1500.01）。根据表4-1，每一次测试都应该设计在条件的边界上。这些条件是各自独立的，我们可以把amount>200和previous_total>1500都设置为false（amount=200.00，previous_total=1500.00）。然而，这个测试将不会消除那些前面测试没能发现的错误，并且不会增加我们对程序正确性的信心。

与逻辑与操作类似，在独立的条件上进行的逻辑或操作也必须用三种情形来测试：第一个条件为true而第二个条件为false；第一个条件为false而第二个条件为true；两个条件均为false。考虑下面的例子，其中displayRelaxationPackage()和displayActivePackage()是在其他地方定义的函数。

```
if (age > 65 || previous_history == 1)
  displayRelaxationPackage();
else
  displayActivePackage();
```

这段代码的测试应该覆盖下面三种情形：

- age > 65为true以及previous_history = = 1为false。
- age > 65为false以及previous_history = = 1为true。
- age > 65以及previous_history = = 1都为false。

开始的两种测试情形经过条件语句的true分支，而最后的测试情形覆盖false分支。由于逻辑操作中的条件是独立的，它们可以同时设置为true。然而，没有必要去测试age>65和previous_history==1都为true的情形，因为这个测试不会消除那些不被前面三个测试所发现的错误。

提示　对于&&操作，用三种情形来测试：第一个条件为false，第二个条件为false，两个条件均为true。对于||操作，也用三种情形来测试：第一个条件为true，第二个条件为true，两个条件均为false。

4.2.3 嵌套条件语句及其优化

嵌套条件语句非常流行。把条件语句放在条件语句的分支中与使用其他类型的语句没有什么不同。向右的缩进排版显示了代码的结构并可以帮助维护人员理解代码设计人员的意图。如果要在分支中写入多于一条的语句，就要对该复合语句使用花括号。使用嵌套条件语句时惟一要警惕的是，要使if和else配对。每一个else应该与最近的if配对：

```
if (condition)
  if (condition1)
    true_statement1;
  else                    // this belongs to the if with condition1
    false_statement1;
else                      // belongs to if (condition)
  if (condition2)
    true_statement2;
  else                    // this belongs to the if with condition2
    false_statement2;
```

这个例子不难理解，因为其中每一个条件语句都是同时具有true分支语句和false分支语句的完整的条件语句。如果缺少其中某一个分支语句的话，情况会变得更加复杂。当程序

员在条件中发现相似性或者试图优化源代码时，也就是使其更加精简而有表现力，这种情况就可能会发生。

让我们考虑一个邮购处理系统中计算订单大小和顾客状况的例子：如果订单数目超过某个小数目（比如，20美元），就不需要服务费；而且，指定的顾客可以获得打折优惠（10%），并要求显示顾客的折扣数。对于小数目订单，任何顾客都没有打折优惠；一般的（非指定的）顾客都要付服务费（每份订单2美元）。这个处理过程的描述显得有点冗长。通常的情况都是这样，因为编写需求的是人，而人的语言不会总是简明扼要的。但是重复的描述通常是会有帮助的，因为它可以防止程序员在解释太精简的文本时产生误解。

程序4-6给出了对以上需求的一种可能的解释。尽管在代码中有3条条件语句，实际上只检查了两种情况（订单的大小和顾客的状况）。由于这些条件各自独立，因此每个条件需要两种测试情形（大订单、小订单、优先顾客、一般顾客）。该程序的运行结果如图4-11至图4-14所示。

程序4-6 嵌套的条件语句

```
#include <iostream>
using namespace std;

int main ()
{
 const double DISCOUNT = 0.1, SMALL_ORDER = 20;
 const double SERVICE_CHARGE = 2.0;
 double orderAmt, totalAmt; int preferred;
 cout << "\nPlease enter the order amount: ";
 cin >> orderAmt;
 cout << "Enter 1 if preferred customer, 0 otherwise: ";
 cin >> preferred;
 if (orderAmt > SMALL_ORDER)
  if (preferred == 1)
   { cout <<"Discount earned " <<orderAmt*DISCOUNT<<endl;
     totalAmt = orderAmt * (1 - DISCOUNT); }
  else
     totalAmt = orderAmt;
 else
  if (preferred == 0)
     totalAmt = orderAmt + SERVICE_CHARGE;
  else
     totalAmt = orderAmt;
 cout << "Total amount: " << totalAmt << endl;
 return 0;
}
```

```
Please enter the order amount: 20
Enter 1 if preferred customer, 0 otherwise: 1
Total amount: 20
```

图4-11 程序4-6的输出结果（小数目，优先顾客）

程序4-6中的实现和需求完全对应，但它的冗余会使很多程序员感到不自在。在不同的分

支里（preferred==1和preferred==0）有着需要优化的相关测试，在不同的分支中（totalAmt=orderAmt）对同一件事有着相似的处理方法。优化这段代码的一种方法是以赋值运算totalAmt=orderAmt开头，接着检查是否因为优先顾客的大订单的打折而需要做修改，或者因为一般顾客的小订单而需要修改服务费。

```
Please enter the order amount: 20.01
Enter 1 if preferred customer, 0 otherwise: 1
Discount earned 2.001
Total amount: 18.009
```

图4-12　程序4-6的输出结果（大数目，优先顾客）

```
Please enter the order amount: 20
Enter 1 if preferred customer, 0 otherwise: 0
Total amount: 22
```

图4-13　程序4-6的输出结果（小数目，一般顾客）

```
Please enter the order amount: 20.01
Enter 1 if preferred customer, 0 otherwise: 0
Total amount: 20.01
```

图4-14　程序4-6的输出结果（大数目，一般顾客）

这种方法使得我们可以消去else子句。程序4-6中的第一种方案可以用下面的伪码描述：

```
if (some_condition_holds_true)
  do_processing_the_first_way;
else
  do_processing_the_second_way;
```

将要实现的优化方案以第二种方式开始进行处理，要么对它进行修改，要么不改变结果。它的伪码如下：

```
do_processing_the_second_way;
if (some_condition_holds_true)
  do_processing_the_first_way;
```

优化方法的具体实现如程序4-7所示。开始两次测试情形的结果与图4-11及图4-12的一样。然而，对于一般顾客的测试结果如图4-15和图4-16所示，它们与图4-13及图4-14所示的结果不同。为什么会这样呢？

程序4-7　优化了的嵌套条件语句

```
#include <iostream>
using namespace std;
```

```
int main ()
{
 const double DISCOUNT = 0.1, SMALL_ORDER = 20;
 const double SERVICE_CHARGE = 2.0;
 double orderAmt, totalAmt; int preferred;
 cout << "\nPlease enter the order amount: ";
 cin >> orderAmt;
 cout << "Enter 1 if preferred customer, 0 otherwise: ";
 cin >> preferred;
 totalAmt = orderAmt;                    // do it the second way
 if (orderAmt > SMALL_ORDER)             // change totalAmt if not a small order
  if (preferred == 1)
   { cout <<"Discount earned " <<orderAmt*DISCOUNT<<endl;
     totalAmt = orderAmt * (1 - DISCOUNT); }
 else                                    // this is an optical illusion
  if (preferred == 0)                    // for small order, check customer status
    totalAmt = orderAmt + SERVICE_CHARGE;
 cout << "Total amount: " << totalAmt << endl;
 return 0;
}
```

```
Please enter the order amount: 20
Enter 1 if preferred customer, 0 otherwise: 0
Total amount: 20
```

图4-15 程序4-7的输出结果（小数目，一般顾客）

```
Please enter the order amount: 20.01
Enter 1 if preferred customer, 0 otherwise: 0
Total amount: 22.01
```

图4-16 程序4-7的输出结果（大数目，一般顾客）

这个实现给我们一种错觉：缩进一定会把设计人员的意图传达给维护人员（和测试人员）。然而，编译程序对此代码的理解方式是不同的。根据else关键字的配对规则，编译程序会把条件语句理解成：

```
totalAmt = orderAmt;                    // do it the second way
if (orderAmt > SMALL_ORDER)             // change totalAmt if not a small order
 if (preferred == 1)
  { cout <<"Discount earned " <<orderAmt*DISCOUNT<<endl;
    totalAmt = orderAmt * (1 - DISCOUNT); }
 else
  if (preferred == 0)
    totalAmt = orderAmt + SERVICE_CHARGE;   // no processing for small orders
```

在这个解决方案中，不管是何种顾客的订单，小数目订单都会不被处理。（小数目订单及优先顾客的结果正确性是偶然的。）对于大订单来说，它被不正确地加上了服务费。注意人的理解和编译程序的理解完全不同，只是看起来好像描述了同一件事情。

在这种情况中，建立一种共同的观点并消除错觉并不太难。所有要做的就是把条件语句

的分支语句都置于花括号中。然而，复合语句不一定非要含有几条语句不可。它可以只含有一条语句。复合语句的标志不是其中所含的语句数目而是代表了块的花括号。程序4-7中的条件语句看起来应该是这样的：

```
totalAmt = orderAmt;                    // do it the second way
if (orderAmt > SMALL_ORDER)             // modify totalAmt if not a small order
  { if (preferred == 1)
    { cout <<"Discount earned " <<orderAmt*DISCOUNT<<endl;
      totalAmt = orderAmt * (1 - DISCOUNT); } }
else
  if (preferred == 0)                   // for small order, check customer status
    { totalAmt = orderAmt + SERVICE_CHARGE; }
```

很多程序员会觉得这种编码风格是有效的，因此他们每次设计条件语句（或者任何其他的控制结构）时都使用花括号。这样有助于避免另一个常见的问题：通常，开始时我们只在条件语句的分支中写了一条语句，因此不需要用到花括号。后来，我们觉得必须在分支语句中加上另一条语句。当我们加上后，有时却忘记了加上花括号，特别是在由维护人员进行改动的时候。在条件语句中的每一个分支语句周围都加上花括号可以减少维护人员在进行改动时必须考虑的事情，这是一个很重要的优点。因此，规范的条件语句看起来应该是这样的：

```
if (expression)
  { true_statement; }         // ready for future expansion
else
  { false_statement; }        // ready for future expansion
```

闰年问题是嵌套条件语句及其优化的另一个好例子。通常，闰年是能被4整除而没有余数的一年。在这里，取模运算符可以发挥很好的作用。作为实现的一部分，我们可以写出如下的内容：

```
if (year%4 != 0)      // if the year cannot by divided by 4, it is not a leap year
  { cout <<"Year " <<year <<" is not a leap year" <<endl; }
else
  { cout << "Year " << year << " is a leap year" << endl; }
```

实际上，这样一个简单的算法已经相当准确。它大约每130年累积误差为1天。因此在这个算法运作了1700年后，日历会增加14天。

因此更准确的计算规则是，如果该年份能被4整除，它是一个闰年；但如果它能被100整除，就不是闰年。我们的代码可以变成：

```
if (year % 4 != 0)              // if year is not divisible by 4, it is not a leap year
  { cout <<"Year " <<year <<" is not a leap year" <<endl; }
else                            // when it is divisible by 4, it is a leap year
  if (year % 100 == 0)          // unless it is divisible by 100
    { cout <<"Year " <<year <<" is not a leap year" <<endl; }
  else
    { cout << "Year " << year << " is a leap year" << endl; }
```

尽管这是事实，但却不是事实的全部。每隔100年这个计算规则就会有1天的误差，并且这个误差太大了。因此，正确的计算规则是，如果该年份能被100整除，它不是闰年，除非该年份能被400整除，它就是闰年。“除非”一词难以从需求转换为代码。这里经常会用到逻辑与运算符（&&）或者嵌套条件。程序4-8给出了实现这个问题的一个程序。如果该年份不能被4整除，那它就不是闰年。如果它能被4以及100整除，它仍不是闰年，除非它能被400整除——这时它才是一个闰年。如果该年份能被4整除但不能被100整除，则它是闰年。系统分

析不是一件容易的事情。

程序4-8 实现闰年问题的一个程序

```
#include <iostream>
using namespace std;

int main ()
{
 int year;
 cout << "Please enter year: ";
 cin >> year;
 if (year % 4 != 0)                          // not divisible by 4, period
  cout <<"Year " << year <<" is not a leap year" << endl;
 else
  if (year % 100 == 0)
   if (year % 400 == 0)                      // divisible by 400 (hence, by 100)
    cout <<"Year " << year <<" is a leap year" << endl;
   else                                      // divisible by 4 and by 100 but not by 400
    cout <<"Year "<<year<<" is not a leap year" <<endl;
   else                                      // divisible by 4 but not divisible by 400
  cout << "Year " << year << " is a leap year" << endl;
 return 0;
}
```

这里有三个条件表达式，因此最坏的情形可能要包含6种测试情形。然而，一些表达式是相关的，因此只需要测试4个分支，即4种测试情形。我们需要测试下列情形：

- year%4!= 0为真（比如1999）
- year%4!= 0为假（也就是，year%4 == 0为真），year%100 == 0为真以及year%400 = = 0（比如，2000）
- year%4 = = 0以及year%100 = = 0均为真，但year%400 = = 0为假（比如，1900）
- year%4 = = 0为真，但year%100 = = 0为假（比如，2004）

图4-17给出了这个代码对2000年的执行结果。这个代码存在很多与其正确性无关，但却与其美观性有关的问题。这里有三层嵌套，很明显要求将条件进行合并。使得year%4==0为真的情形对于闰年问题有两个分支，它们也应该合并起来。怎样做呢?首先，将条件否定，以便使相似的分支更加相互接近。例如：

```
if (year % 4 != 0)          // not divisible by 4, end of story
 cout <<"Year " << year <<" is not a leap year" << endl;
else
 if (year % 100 == 0)
  if (year % 400 != 0)      // divisible by 100 but not by 400)
   cout <<"Year "<<year<<" is not a leap year" <<endl;
  else                      // divisible by 4, by 100 and by 400
   cout <<"Year " << year <<" is a leap year" << endl;
 else                       // divisible by 4 but not divisible by 100
  cout << "Year " << year << " is a leap year" << endl;
```

现在，可以使用与运算符把相邻的两个条件合并在一起，而且对于闰年的最后两个子句也可以合并在一起。于是得到一个更加精简的程序：

```
if (year % 4 != 0)                          // not divisible by 4, period
  cout <<"Year " << year <<" is not a leap year" << endl;
else
  if (year%100==0 && year%400!=0)    // by 100 but not by 400
      cout <<"Year "<<year<<" is not a leap year" <<endl;
  else                                      // divisible by 4 but not divisible by 100
    cout << "Year " << year << " is a leap year" << endl;
```

```
Please enter year: 2000
Year 2000 is a leap year
```

图4-17　程序4-8的输出结果（年份可以被4、100、400整除）

这不是很好吗？这里只有两层嵌套，并且相当容易理解。但重复了关于非闰年的相同处理，这对于C++程序员来说还是不够好。当年份不被4整除或者当条件（year%100==0和year%400!=0）为真时，该年份就不是闰年。这就要求使用逻辑或操作，程序4-9不仅正确和有效，而且精简又美观。

程序4-9　实现闰年问题的一个优化的程序

```
#include <iostream>
using namespace std;

int main ()
{
  int year;
  cout << "Please enter year: ";
  cin >> year;
  if (year % 4 != 0 || year % 100 == 0 && year % 400 != 0)
    cout <<"Year " << year << " is not a leap year" <<endl;
  else
    cout << "Year " << year << " is a leap year" << endl;
  return 0;
}
```

用刚才讨论的测试情形来运行该程序可以产生和程序4-8一样的结果（如图4-17所示）。

毫无疑问，程序4-9中的程序比程序4-8中的程序好。然而，为了消除代码中的多余的6行（并证明它是正确的）所花的时间是否值得，这是一个有争议的问题。

有时，当花费了几个小时优化了复杂的条件语句时，我们会对结果感到骄傲。这些努力是否经济，由程序员自己来决定。

4.3　循环

在程序设计中，条件语句扮演了一个很重要的角色。它们是每一个程序的主力，但只有它们还不能完成所有的任务。在每一个程序中，很可能为了不同的顾客、事务、在线用户等的需要，要重复同样的语句序列。这些工作就需要有循环结构。

对于重复的动作，C++提供了3种循环语句：while循环、do-while循环和for循环。C++的每一个循环控制着单个语句（它以分号结束）或一条位于花括号里的复合语句（块）

（在块的闭括号后没有分号）的重复。为了控制循环，所有循环都要用到类似于条件语句中的逻辑表达式。这些逻辑条件的值为true或false（非零或0）。每一次循环都会测试它们，当循环条件变为false时，该循环就会终止；如果条件为true，循环体（语句或块）就重复执行。注意，这里是“每一次循环”而不是“在每一次循环之前”，因为不同循环的循环条件的测试方式是不同的。不管循环如何设计，循环体都必须有某些动作能改变循环的条件。否则，循环条件可能永远为true——这对于用到循环的程序来说就是一个威胁。

while循环在通过循环体的每一次循环之前测试其循环条件，且当循环条件变为false时就停止循环。循环表达式在第一次循环之前测试。因此，循环体可能重复0次——如果首次进入循环时其循环条件为false。

do-while循环在每一次循环之后测试其循环条件，且当条件变为false时停止循环。由于条件的第一次测试是在第一次循环之后而不是在其之前进行，因此循环至少进行一次。

for循环通常用在预先能确定重复次数的情形。

通常，同样的算法可以用任何一种循环格式来设计，而其选择只是个人品味而已。有时，某种格式会显得比其他的更好，因为所需要的语句更少，或者与其他语句更匹配。

4.3.1 while循环

while循环被作为一个单独的语句来执行；它和其他语句的不同之处是其循环体会根据循环的逻辑条件的值来决定是否重复地执行。

while循环具有如下的逻辑结构：

```
previous_statement;
while (expression)          // this is the loop expression
  statement;                // this is the loop body
next_statement;
```

当逻辑条件（表达式）为true时，控制结构就重复执行其循环体。最终（或甚至是在第一次通过之前）当条件变为false时，就会跳过循环语句而执行next_statement。当循环体有若干条语句时，要使用到块作用域定界符（花括号）。

```
while (expression)
  { statement;              // notice the indentation
    . . .
    statement; }            // end of loop body
```

设计循环的一个常见的错误是：其循环体没有把表达式的值变为false；程序会在一个“无限循环”中继续执行，因而只能在操作系统的帮助下才能终止程序。

通常，循环的设计围绕着一个所谓当前数据的概念。我们重复地处理着某些数据项。这意味着数据项必须被初始化、求值和做相应的处理（打印、用于计算、保存或算法所需的其他任何处理），随后，它必须为了进行下一次循环而被修改、求值和处理，直到处理到最后一项为止。这些处理可以用下列的模式合并为while循环结构：

```
initialize_current_data;
while (evaluate_current_data)       // decision point
  { process_current_data;           // main goal of this code
    change_current_data; }          // do not forget this step!
```

让我们考察一个事务处理的例子。为了简单起见，假定程序输入5个数据并对它们进行求和（将使之更理想化）。由于已知事务的总额，因此可以用一个变量来存放已经处理的事务数目。这是对当前数据的一部分操作：它必须初始化为0并在每次事务之后加上1。当前数据的另一项是输入的总数量，它也必须初始化为0并在每次循环时把事务数量加到总数量中。然而，不应该用总数量来检查循环是否应该终止，而应该用事务数目进行检查。因此，循环的步骤如下：

- initialize_current_data：count设置为1，total设置为0。
- evaluate_current_data：测试事务数目是否超过5。
- process_current_data：输入下一个事务数量，把它加到total中。
- change_current_data：使事务数目加1，并测试它。

程序4-10给出了实现该设计的程序。这个为特定大小的数据集而编写的程序不大实际。

程序4-10　一个无限次循环的while循环程序

```
#include <iostream>
using namespace std;

int main ()
{
 double total, amount; int count;
 total = 0.0; count = 1;                    // initialize current data
 while (count <= 5)                         // evaluate current data
  { cout << "Enter the amount: ";
    cin >> amount;                          // enter current data
    total += amount; }                      // process current data
 cout<< "\nTotal of 5 transactions is "<<total << endl;
 return 0;
}
```

这是一个典型的程序设计错误的例子。因为只要count没有超过5，循环就一直执行。当count到达值6时，循环应该终止。问题是count将永远不会到达6（或任何其他值），因为循环体中没有改变count的值。为了改正这个情况，循环体在每次循环时应该把1加到count中。在循环体的最后这样做是合适的：

```
while (count <= 5)          // evaluate current data
 { cout << "Enter the amount: ";
   cin >> amount;           // enter current data
   total += amount;         // process current data
   count++; }               // change current data: do not forget it!
```

我们希望写出这样的程序：代码段（比如，处理一个事务）的使用次数与每个输入数据所需的次数一样多，并且数据集的大小对于程序的不同次运行是不同的，像程序4-10那样强硬地规定数据集的大小是不合适的。程序要知道什么时候处理到了数据集的最后一个元素。解决这个问题的一个方法是直接询问用户要处理多少项，并在循环的条件中用这个值作为限制。然而，并不总是依赖用户来输入数据。例如，可以从一个远程的计算机通过一条通信线来输入数据。在这种情形下，第一项数据通常就是要处理的数据项的数目，但数据集的大小可能无法预先知道，或者可能会很大。对5个数据项计数是一回事，对几百个或几千个数据项计数又是另一回事。

重复处理的一个更常见的方法是，逐项输入数据并询问用户（或通过输入文件或通过一条通信线）是否还有另一项要处理。

要完成这个任务通常有两种方式。一种方式是用不同的变量来存放事务数据，并且由用户回答是否还有更多的数据。在输入每一个事务后，用户就要回答是否还有更多的项目要处理。第二种方法是用户通过输入一个特殊的值（称为岗哨值），以便告诉应用程序该数据集输入结束了。一个岗哨值是一个特殊的值，它和一般的数据不一样，它本身不是有效数据，只是用来表示有效数据的结束，对于事务数量来说，可用一个负数或0作为岗哨值。当数据通过通信线传送时，岗哨值是数据集的最后一个值。

对于事务数量，这样的一个特殊值可以是0或一个负数。程序4-11给出了使用一个负数或0作为岗哨值的循环程序。

程序4-11 用一个负数或0作为标记来实现while循环

```
#include <iostream>
using namespace std;

int main ()
{
 double total, amount; int count;
 total = 0.0; count = 0;                    // different initialization
 amount = 1.0;                              // an artificial trick: why 1 and not 10?
 while (amount > 0)                         // evaluate current data
  { cout << "Enter amount (negative or zero to end): ";
    cin >> amount;                          // enter current data
    total += amount;                        // process current data
    count++; }
 cout << "\nTotal of " << count << " transactions is "
    << total << endl;
 return 0;
}
```

在程序4-10中，只要count没有超过5，循环就继续进行。在第一次循环之前count是1，在第二次循环之前它是2，在第五次循环之前它是5，而在第五次循环之后它是6，于是count<=5变为false。这对程序4-11来说没有什么作用，因为在运行结束时我们想利用count的值来反映已处理的事务数目。因此在程序4-11中count被初始化为0而不是1。

以下就是C++程序员（实际上，任何程序员）在建立循环时都应该考虑到的有关问题：循环的初始值是正确的吗？它们能确保正确的终止值吗？因此在这个例子中作者只使用了一个很小的数值，以便更容易地跟踪循环。程序4-11的样本运行结果如图4-18所示。

```
Enter amount (negative or zero to end): 22
Enter amount (negative or zero to end): 33
Enter amount (negative or zero to end): 44
Enter amount (negative or zero to end): -1

Total of 4 transactions is 98
```

图4-18 使用一个负数或0作为岗哨值的程序4-11的输出结果

从结果可见事务数目是不正确的！在用户输入-1.0之前，count的值是3，这是正确的。

但在用户输入-1.0之后，count变为4，这是不正确的。更加糟糕的是，这个负数也被加到了total中。因此，total的值也是不正确的。

这个技术问题有很多解决方法。我们可以把count初始化为-1，或者我们可以在循环之后将count减1，也可以对总数total执行同样的操作。在循环之后的代码可以这样写：

```
count--; total -= amount;              // after-loop correction
cout << "\nTotal of " << count << " transactions is "
     << total << endl;
```

这个解决方法虽然不优美，但可行。另一个解决方法是在循环的中间加上一个条件语句，并且只有当amount的值不是岗哨时才改变total和count的值：

```
while (amount > 0)                     // evaluate current data
  { cout << "Enter amount (negative or zero to end): ";
    cin >> amount;                     // enter current data
    if (amount > 0)                    // test for end of data
     { total += amount;                // process current data
       count++; } }
```

可以看到所有这些都是以代码的复杂度为代价来解决问题的，它们并不优美。通常（但不总是），这反映了一个概念性的问题。实际上，程序4-11中的程序还有另一个概念性而不是技术性的问题，这就是第一次通过循环时的问题。当C++变量分配了内存空间之后，它含有随机数（这不是全部的事实，我们以后将讨论它）。这一随机数在程序的某些运行测试中可能是0。这就意味着程序还没有处理输入数据就会终止。为了避免这种情况发生，要把amount初始化为某个值，其目的是防止循环过早地终止。

从软件工程的观点来说，这是不正确的。在应用环境中所用的1.0没有什么语义含义。如果是2.0，结果还会一样。这个值没有表达出设计的任何意图，因此妨碍了设计人员与维护人员之间的交流。在意识到它没有任何含义之前，维护人员将要花时间去搞清楚这个1.0的含义是什么。或许这不用花费很多时间；然而，正是这种小毛病使得应用程序增加了不必要的复杂度。

这个循环的另一个问题是，它没有正确地处理岗哨值。当用户输入该负数（或一个负数通过一条通信线达到其传输的终点）时，这个值首先被加到total中，随后才被用来终止循环。

解决这些问题的方法是改变循环体的结构。在程序4-11中，循环体首先接受amount（它可能是一个岗哨值），然后处理它。作者的建议是首先处理在前一次循环中已输入的amount值，并且只是在循环体的尾部才为下次循环而接受amount。该循环结构应该是：

```
while (amount > 0)                     // evaluate current data
  { total += amount;                   // process current data
    count++;
    cout << "Enter amount (negative or zero to end): ";
    cin >> amount; }                   // change current data
```

但第一次循环之后的结果如何呢？在程序4-11中所使用的1.0是不合适的，是否应该把变量amount初始化为0呢？将0加到total中是无害的，但它将使第一次测试时终止循环——因为条件amount>0的值为false。另外，如果我们需要打印输入的数量或用其他特殊的方法来处理它，这个解决方法也将不起作用。

这里一个更好的解决方法是所谓的预读（prime read）技术。在循环之前接受第一个值，

在循环的头部处理该值，然后在循环的尾部接受下一个值，并在下一次循环的头部处理它。在程序4-12中可以看到这种解决方法，其测试运行的结果如图4-19所示。

程序4-12 用预读来实现的while循环

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 0;           // different initialization
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount;                    // enter current data (first time)
  while (amount > 0)                // evaluate current data
  { total += amount;                // process current data
    count++;
    cout << "Enter amount (negative or zero to end): ";
    cin >> amount; }                // change current data
  cout << "\nTotal of " << count << " transactions is "
       << total << endl;
  return 0;
}
```

```
Enter amount (negative or zero to end): 22
Enter amount (negative or zero to end): 33
Enter amount (negative or zero to end): 44
Enter amount (negative or zero to end): -1

Total of 3 transactions is 99
```

图4-19 用预读之后的程序4-12的输出结果

现在所有的问题（包括复杂性）都已解决了。变量count和total被初始化为逻辑初始值（0）。变量amount从不被初始化，因为无论我们给它赋予什么值，都不会用到它的初值，它的值会被输入操作重写。在循环之后不会对已被循环不止确修改的值进行调整。

这个解决方法的缺点是把输入语句编码了两次。在实际工作中，这并不是什么大事件，因为输入和验证输入数据都要用到两条以上的语句，并且可能会封装在一个函数中，因此我们必须两次调用输入函数，这是可以接受的。没有什么事是完美的。

在继续讨论do-while循环之前，还要说明while循环设计的其他一些问题。这里以字符流的处理作为一个例子。为了简单起见，所做的处理是回显字符、计算总字符个数以及空格的数目。在任何情况下，该处理将继续执行，直到用户按下Enter键（字符'\n'）为止。我们将采用有预读的循环结构，第一个字符在while循环之前输入，在循环的头部回显以前已输入的字符，计算字符的数目并检查它是否是一个空格字符。在循环的尾部，读入下一个字符。循环的条件检查这一个字符是否是换行符，如果不是（循环条件为true），处理就继续进行，循环的头部回显字符并对该字符计数，然后循环的尾部接受下一个字符。如果它是换行符，循环条件的值变为false，于是循环终止。

为了输入字符，使用了iostream库中的函数get()，把它当做一个消息传给cin对象。我们已经在第2章的2.7节"类"中讨论过消息的语法，因此继续使用它是很好的。实际上，

为了正确使用它，我们只需了解函数调用cin.get　)可以从输入缓冲区中返回下一个字符。

程序4-13给出了实现这个问题的程序，而图4-20给出了测试运行的结果。

程序4-13　使用预读输入字符的while循环

```
#include <iostream>
using namespace std;

int main ()
{
 char ch; int count = 0, spaces = 0;       // initialize counters
 cout << "\nType a sentence, press Enter\n";
 ch = cin.get();                           // prime read for the loop
 while (ch != '\n')                        // no semicolon after the condition
  { cout << ch;                            // process data: echo, check, count
    if (ch == ' ')
     spaces++;
    count++;
    ch = cin.get(); }                      // change current data
 cout << "\nTotal number of characters " << count << endl;
 cout << "Number of spaces is " << spaces << endl;
 return 0;
}
```

```
Type a sentence, press Enter
This is a test.
This is a test.
Total number of characters 15
Number of spaces is 3
```

图4-20　程序4-13的运行结果（处理输入字符）

在这里我们又遇到了表面现象与实际不同的情况。代码表示用户输入的第一个字符在用户输入第二个字符之前显示，第二个字符在用户输入第三个字符之前显示，等等。如果运行这个程序，将会看到这些字符直到按下Enter键后才一起显示。其原因是cin.get()调用不是从键盘而是从计算机内存中的一个内部缓冲区读入数据，当用户按下键盘的某个键时，数据被传到缓冲区中；只有在用户按下Enter键或缓冲区满了时，数据才可用于程序。当程序需要频繁的文件I/O操作并且每一次都是存取少量数据时，使用缓冲区可以提高程序的性能。有了缓冲区，对于大量的数据，缓慢的外部文件I/O只需进行一次。（它几乎占用同样长的时间，与数据的数量无关。）这样，直接作用于内存缓冲区的多次I/O操作在速度上会快得多。（对于这个程序这并没有太大的作用。）因此，如果在输入数据时没有看到其输出，请不要惊讶。

注意，在语句ch=cin.get()之后就立即测试循环条件while(ch!=\n)；第一次测试是在循环的前一条语句；其他测试是在循环体尾部的那条语句。这个代码结构代表了一种流行的C++用法：把赋值运算和条件测试结合起来。程序4-14给出了这个重要的用法。

程序4-14　循环条件中带有赋值运算的while循环

```
#include <iostream>
using namespace std;
```

```
int main ()
{
  char ch; int count = 0, spaces = 0;        // initialize counters
  cout << "\nType a sentence, press Enter\n";
  while ((ch = cin.get()) != '\n')           // change current data
  { cout << ch;                              // process next symbol
    if (ch == ' ') spaces++;                 // OK for a single line
    count++; }
  cout << "\nTotal number of characters " << count << endl;
  cout << "Number of spaces is " << spaces << endl;
  return 0;
}
```

这是一种非常流行的C++用法。但不能把它用在程序4-12中，因为那里的输入语句（cin>>amount;）不会返回用户输入的值。实际上，它确实返回了一个值，但这是cin对象的值而不是用户输入的值。程序4-13中的输入语句（ch=cin.get();）能够返回用户输入的字符值，并能够用在程序4-14的循环条件中。

请注意程序4-14中输入语句两边的圆括号。省略了它们将不会出现语法错误，但它改变了代码的含义。

```
cout << "\nType a sentence, press Enter\n";
while (ch = cin.get() != '\n')       // no parentheses around input statement
{ cout << ch;
  if (ch == ' ')
    spaces++;                        // process next symbol
  count++; }
```

在这里，运算符的优先级十分重要。赋值运算符的优先级低于不相等比较的优先级。这就意味着编译程序把代码理解成：

```
while (ch = (cin.get() != '\n'))     // quite a different story
```

输入字符以后，才把它与换行字符进行比较。对于所有的字符（除了输入中的最后一个字符以外），比较的结果都为true，它们都不是换行符，并且变量ch得到的值为1（它不是一个可打印码）。这样显示的字符是不正确的，且其空格数将为0。

不要抱怨C++以错误的顺序执行运算。应该记住运算的顺序，避免这些问题的发生。如果有怀疑（即使没有怀疑），可以使用圆括号。

4.3.2 do-while循环

do-while循环非常类似于while循环。通常，它们可以互相转换地使用。两者的主要差别是，do-while循环是每一次循环之后在循环体的尾部测试循环条件，而while循环则是每次循环之前在循环体的头部测试它的条件。

类似于while循环，do-while循环控制是由一条单语句（或位于花括号里的块语句）组成的重复执行的循环体。do-while循环的一般结构为：

```
previous_statement;
do
  statement;                 // or { statements }
while (expression);
next_statement;
```

在previous_statement执行之后，关键字do之后的循环体就被执行。执行完之后，对循环表达式求值，如果它的值为true，则循环体又继续执行。如果表达式为false，该循环就终止，然后执行next_statement。

这种结构保证了循环体至少被执行一次。为了终止循环和避免无限循环，循环体必须能够最终改变循环表达式的值。

为了防止混淆，程序员经常使用花括号，即使循环体只有一条单语句，形如：

```
do
  { statement; }
while (expression);
```

在do-while循环中，必须在循环表达式后面放置一个分号以避免语法错误。这恰好和while循环中的情形相反，在那里，不能在循环条件之后放置分号。请注意，在while循环的循环条件后放置一个分号并不会使编译程序感到困惑并发出语法错误的信息，但却会产生不正确的代码（语义错误）。为了向维护人员表明这个while关键字是特殊的，必须在该行的结尾处使用分号，有些程序员把闭花括号和关键字while写在同一行：

```
do
  { statement;
} while (expression);        // the brace warns about the presence of the semicolon
```

该循环体的设计方法与while循环类似。

```
initialize_current_data;
do { change_current_data;
     process_current_data;
      } while (evaluate_current_data);
```

对于事务处理和计算总数的例子，这种结构的组件必须完成下列操作：

```
initialize_current_data:  set total and count to zero
change_current_data:      enter new value of amount
process_current_data:     if positive, increment total and count
evaluate_current_data:    test if the amount is a sentinel
```

程序4-15给出了实现这个版本的程序。这个版本的输出当然与程序4-12输出的图4-19是一样的。

程序4-15　没有预读的do-while循环

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 0;              // initialize current data
do {
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount;                       // enter (change) current data
  if (amount > 0)                      // check for end of data
  { total += amount;                   // process current data
```

```
    count++; }
  } while (amount > 0);                    // evaluate current data
  cout << "\nTotal of " << count << " transactions is "
       << total << endl;
  return 0;
}
```

我们可以看到，一方面do-while循环简化了初始化操作，并消除了预读的需要（与程序4-12比较）；另一方面，它需要在循环体的中间增加一条额外的条件语句，以避免把岗哨值错误地看做是一个合法的输入值。

对输入空格字符的计数可以在程序4-16中的do-while循环中进行。do-while循环的使用消除了预读的需要。与程序4-15的例子相似，这个结构要求检查其循环体，以便检验岗哨值是否已输入。因此，对当前数据求值了两次：第一次是在循环体中，第二次是在循环条件中。

程序4-16 实现字符输入的do-while循环

```
#include <iostream>
using namespace std;

int main ()
{
  char ch; int count = 0, spaces = 0;      // initialize data
  cout << "\nType a sentence, press Enter\n";
do {
    ch = cin.get();                        // change current data
    if (ch != '\n')                        // check for end of data
     { cout << ch;
       if (ch == ' ') spaces++;            // process current data
       count++; }
     } while (ch != '\n');                 // evaluate current data
  cout << "\nTotal number of characters " << count << endl;
  cout << "Number of spaces is " << spaces << endl;
  return 0;
}
```

在这里，字符ch的值通过赋值确定，并且很快地在条件语句中被测试，从而有机会把赋值以及测试结合在一条语句中，如程序4-17所示。

程序4-17 在条件语句中带有赋值运算的do-while循环

```
#include <iostream>
using namespace std;

int main ()
{
  char ch; int count = 0, spaces = 0;      // initialize data
  cout << "\nType a sentence, press Enter\n";
  do {
    if ((ch = cin.get())!= '\n')           // change current data
     { cout << ch;
       if (ch == ' ') spaces++;            // process current data
       count++; }
     } while (ch != '\n');                 // evaluate current data
```

```
  cout << "\nTotal number of characters " << count << endl;
  cout << "Number of spaces is " << spaces << endl;
  return 0;
}
```

同样，这个优化并没有改变程序的性能和正确性，但它使代码变得更加精简和美观。

4.3.3 for循环

如果在循环开始之前可以预先知道重复的次数，那么通常使用for循环比较合适，这不是一个重要的因素。这种形式的循环从视觉上把循环设计的三个最重要的元素放在了一起：在第一次循环之前对当前值初始化、在开始下次循环之前对当前值求值以及在（本次）循环之后（在下次循环之前）改变当前值。在其他循环结构中，这些元素被分散地写在循环的不同位置上。

for语句具有以下的标准形式，这里我们把控制循环体执行的三个表达式合并在圆括号里（当前值的初始化、对它的求值和对它的修改）。注意这些表达式要用分号隔开。它们只是被分隔开，而不是被终止（像语句那样），因此在闭括号之前的最后一个表达式后面没有分号。

```
previous_statement;
for (InitialExpr; ContinuingExpr; IncrementExpr)
  statement;                    // compound statement in braces is OK
next_statement;
```

InitialExpr只在第一次循环之前求值一次。这里是一个为了建立循环而对索引、计数器、总量等数值进行初始化的方便之处。

IncrementExpr在每次执行循环体之后立即求值。这是一个修改当前值、增大索引、计数器等数值的方便之处。

ContinuingExpr在第一次循环之前和每次循环之前求值。这个表达式计算是否有需要执行下一次循环，如果这个表达式的值为true，循环语句就被执行，而且在这之后对IncrementExpr立即求值；随后，又对ContinuingExpr求值以决定是否有必要执行下一次循环。如果这个表达式的值为false，循环就终止。

请注意for循环和下面的while循环是等价的。

```
previous_statement;
InitialExpr;
while (ContinuingExpr)
  { statement;                  // or a sequence of statements
    IncrementExpr; }
next_statement;
```

程序4-18给出了使用for循环实现的事务处理程序。其初始化包括把count设置为0；其继续循环测试包括对岗哨值的测试（因此仍然需要预读）；其增量表达式包括使变量count加1的工作。

程序4-18　用for循环实现事务处理的程序

```
#include <iostream>
using namespace std;
```

```
int main ()
{
 double total, amount; int count;
 total = 0.0;                              // different initialization
 cout << "Enter amount (negative or zero to end): ";
 cin >> amount;                            // enter current data
 for (count=0; amount>0; count++)          // three expressions
 { total += amount;                        // process current data
   cout << "Enter amount (negative or zero to end): ";
   cin >> amount; }                        // change current data
 cout << "\nTotal of " << count << " transactions is "
      << total << endl;
 return 0;
}
```

for循环的三个表达式中的每一个都是一个表达式，这种“深奥的”观察意味着用逗号分隔开的一系列表达式可以作为这些表达式中的每一个表达式。请记住，逗号在C++中是一个运算符：它从左向右对其操作数进行求值并返回最右的那个值。在for循环中，返回值是不重要的，没有必要要它们。惟一的例外是决定下一次循环是否应该执行的Continuing Expr。这意味着我们可以像程序4-19那样扩展InitialExpr。

程序4-19 在初始化表达式中使用逗号运算符的for循环

```
#include <iostream>
using namespace std;

int main ()
{
 double total, amount; int count;           // no initialization
 cout << "Enter amount (negative or zero to end): ";
 cin >> amount;                             // enter current data
 for (total=0.0, count=0; amount>0; count++)
 { total+=amount;
   cout << "Enter amount (negative or zero to end): ";
   cin >> amount; }                         // change current data
 cout << "\nTotal of " << count << " transactions is "
      << total << endl;
 return 0;
}
```

一个更加有趣的表达形式是程序4-20中的字符处理示例。在这里，初始化表达式中使用了逗号隔开的表达式，赋值运算被当做比较的一部分继续用在循环表达式中（请将这个版本与程序4-13及程序4-16进行比较）。

程序4-20 在继续循环表达式中使用赋值运算的for循环

```
#include <iostream>
using namespace std;

int main ()
{
 char ch; int count, spaces;            // no initialization
 cout << "\nType a sentence, press Enter\n";
 for (count=0, spaces=0; (ch=cin.get())!='\n'; count++)
  { cout << ch;                         // process next input symbol
```

```
    if (ch == ' ') spaces++; }
  cout << "\nTotal number of characters " << count << endl;
  cout << "Number of spaces is " << spaces << endl;
  return 0;
}
```

在一个for循环中，我们可以做的另一件有趣的事情是在循环的InitialExpr中定义变量。例如，考察一下程序4-21。该程序实现前面的自然数的平方数求和，要求和的平方数的数目由用户输入。for循环把变量n初始化为1，测试n的值是否到达限定值num，并在每次循环后使n加1。由于变量n只用在循环中，因此没有必要在一个更大的作用域中定义它。因此这个变量只需定义在for语句中而不必定义在main()函数里。这是一个流行的C++用法，这个程序的测试运行结果如图4-21所示。

程序4-21　使用for循环计算平方数的总和

```
#include <iostream>
using namespace std;

int main ()
{
  int sum=0, num;
  cout << "\nEnter the number of squares to add: ";
  cin >> num;
  for (int n = 1; n <= num; n++)
    { sum += n * n; }
  cout << "Total of squares is " << sum << endl;
  return 0;
}
```

```
Enter the number of squares to add: 4
Total of squares is 30
```

图4-21　程序4-21的运行结果（自然数的平方相加）

从程序4-19和程序4-20中可以看到，C++允许程序员在for循环的初始化表达式中初始化若干个变量。C++也允许在初始化表达式中定义若干个有着相同类型的变量。在程序4-22中，变量n和变量sum都是在循环中定义的。另外，变量sum在使用逗号运算符的增量表达式中被更新。这个版本的输出结果与程序4-21的一样。正如大家所见到的，这时循环体退化为一条空语句。

程序4-22　退化为一条空语句的for循环

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout << "\nEnter the number of squares to add: ";
  cin >> num;
  for (int sum = 0, n = 1; n <= num; sum+=n*n, n++);    // !!
```

```
    cout << "Total of squares is " << sum << endl;
    return 0;
}
```

很多程序员不喜欢在循环语句的结尾处使用分号。这不是放置分号的常见地方，它可能会使维护人员感到迷惑。为了把设计人员的知识更好地传给维护人员，这些程序员把分号放在单独的一行，例如：

```
for (int sum = 0, n = 1; n <= num; sum+=n*n, n++)
  ;                                                    // !!
```

有些程序员完全不用空语句，因为它们太容易令人困惑；而是使用类似于程序4-21中的结构，使得其循环体中至少有一条语句。程序4-21的最大问题是可移植性问题。变量num在循环中定义，但却在循环终止之后继续使用。早期的C++允许这种用法，然而，新的标准C++把它当做是一个语法错误；程序员只能在循环中定义那些不会用在循环之外的变量。大多数的编译程序都会让这个程序编译通过，但仍不应该使用这种代码。不要过度地优化for语句，这可能是一个好的建议。

4.4 C++转移语句

条件语句和循环语句是程序设计中必不可少的工具。不使用它们，我们甚至不能编写出一个最简单的程序。它们是必要的。其他的控制语句有用但不是必要的。它们的语法形式使得我们的程序更加精简和美观。

其他的C++控制语句包括不同种类的转移语句。程序设计人员喜欢使用转移，因为它允许程序员有效地把控制转移到程序源代码的任何地方。然而，使用转移的程序比没有使用转移的程序更加难以分析。当控制流是顺序的时候，为了理解一条程序语句的执行结果，维护人员只需要理解正在分析的语句之前的那些语句。当程序控制可以从程序的不同地方转移到某条语句时，所有这些地方都可以影响这条语句的工作。这使得维护人员的工作更加困难。因此在程序设计中，转移的声誉并不好。

C++试图采用折中的方法。它允许转移，以便程序员能够编写出精简而且功能强大的代码。另一方面，它又限制转移，以便维护人员不需要完成太难的任务。

4.4.1 break语句

break语句用于从一个循环中立即退出；在执行了这条语句之后，控制流转移到循环语句之后的那条语句。它可以与while、for以及do-while循环一起使用，在循环的中间放弃循环，不会导致控制流太令人困惑。然而，break语句不能用于从if语句的一个分支中转移出来，否则将导致控制流太令人费解。在本节的后面，我们将会看到在switch语句中使用break语句的情形。

无条件地执行break语句没有多大的意义，它意味着没有执行任何循环。break语句通常在条件语句中使用，这个条件的逻辑表达式指定了循环终止的条件。通常，这样可以简化循环的条件；例如，循环可能被设置为“永远”地执行下去。

例如，考察一下程序4-12中的循环，它处理输入数据，直到岗哨值出现时才停止。循环条件使用了变量amount的值，因此这个变量必须在该循环之前初始化，并且这就是预读要做

的工作。

```
cout << "Enter amount (negative or zero to end): ";
cin >> amount;                                  // enter current data
while (amount > 0)                              // evaluate current data
{ total += amount;                              // process current data
  count++;
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount; }                              // change current data
```

使用break语句允许我们使用某些不变的东西去代替循环条件，例如，while(1==1)。由于这个条件总是为true，书写这个表达式就会很少犯错误。由于这个条件没有使用变量amount的值，因此没有必要初始化这个值。因此，可以消除预读，并且可以在循环的头部而不是在尾部接受amount的新值。这个循环结构的问题是当岗哨值出现时怎样终止循环。break语句提供了解决方法。如果想在amount>0时继续循环，那么循环的终止条件是该表达式的否定，也就是amount<=0.0。当这个（否定的）条件为true时，就执行break语句，然后控制流离开循环而转移到下一条语句。

```
while (1 == 1)                                  // loop forever
{ cout << "Enter amount (negative or zero to end): ";
  cin >> amount;                                // change current data
  if (amount <= 0.0) break;                     // explicit break
  total += amount;                              // process current data
  count++; }
```

有人可能认为，把测试表达式amount>0从循环表达式转移到break语句，并没有对代码的复杂度有多大的改进，但请注意它使得我们可以消去预读操作。

这个算法的do-while版本（见程序4-15）没有使用预读，但它必须对下一个输入值是否为合法值而检查两次，分别在循环的中间以及在循环的条件中：

```
do {
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount;                        // enter (change) current data
  if (amount > 0)                       // evaluate current data
  { total += amount;                    // process current data
    count++; }
  } while (amount > 0);                 // evaluate current data again
```

break语句允许用总是为真的简单条件来代替循环条件（比如，1==1，甚至是1）。为了在岗哨值出现时终止循环，条件amount>0必须像前面那个例子一样取否定。当amount<=0时，break语句就把控制转移到该循环之后的那条语句。该循环结构从某种程度上被简化了，不再需要有带花括号的局部复合语句。注意，由于在C++中将非0值作为true，因此循环while(1)是while(1==1)的一个合法的替换形式。

```
do {
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount;              // enter (change) current data
  if (amount <= 0) break;     // evaluate current data
  total += amount;            // process current data
  count++;                    // no need for compound statement
} while (1);                  // no need to evaluate current data here
```

使用break语句的另一个例子是范围检查，它经常用于检查输入数据的有效性。例如，假设用户必须输入从1到5范围内的响应值。如果用户输入错误，就必须重新输入，直到输入

值合法为止。在程序4-23中使用了do-while循环，因此循环体至少被执行一次。如果输入值无效，变量error_flag被设置为1；如果输入了有效范围之内的值，则error_flag被设置为0。

程序4-23 使用do-while循环检查输入的有效性

```
#include <iostream>
using namespace std;
const int N = 5;

int main ()
{
 int num, error_flag;
 do {
  cout << "Enter number between 1 and " << N << ": ";
  cin >> num;
  if (num < 1 || num > N)
   { cout << "This is incorrect; please repeat.\n";
     error_flag = 1; }
  else
   error_flag = 0;
 } while (error_flag == 1);
 cout << "Your input is " << num << endl;
 return 0;
}
```

这是实现程序不同部分之间通信的一种常用技术。在程序的某个地方（循环条件）想知道在程序的另一个地方（循环体）发生了什么。为此，我们在程序的某个地方测试在程序的另一地方所设置的变量。

有些程序员不喜欢增加标记以及其他控制变量，以实现从程序的一个地方把信息传送到另一个地方，因为这会增大代码的耦合度和复杂度。这个算法的另一种实现方法是重复在循环条件中的测试而不使用出错标记。

```
do {
 cout << "Enter number between 1 and " << N << ": ";
 cin >> num;
 if (num < 1 || num > N)
  cout << "This is incorrect; please repeat.\n";
 } while (num < 1 || num > N);
```

这是一个更加精简的解决方法。而另一个方法是使用无限循环并且当取值合法时中断循环，由于当N<1或num>N时不断循环地要求输入数据，因此当条件变为false时就会终止循环。

```
do {
 cout << "Enter number between 1 and " << N << ": ";
 cin >> num;
 if (!(num < 1 || num > N)) break;
 cout << "This is incorrect; please repeat.\n";
 } while (true);
```

注意这是“无限”循环的第三种形式，它用true的值作为循环的条件。

很多程序员喜欢显式地否定复合条件。为了这样做，可以用||运算符代替每一个&&运算符、用&&运算符来代替每一个||运算符，并对每一个单独的条件进行否定。例如，考虑表达式a1&&(~a2)||a3，其中a1、a2和a3都是布尔表达式，它的否定是(~a1)||a2&&(~a3)。

在以上的例子中，num<1的否定是num>=1，而num>N的否定是num<=N。现在终止中断的条件是：

```
do {
  cout << "Enter number between 1 and " << N << " : ";
  cin >> num;
  if (num >= 1 && num <=N) break;                    // nice and simple
  cout << "This is incorrect; please repeat.\n";
  } while (true);
```

有些程序员会对复合条件的否定感到不熟悉；但这是一个很有用的技术，应该尽可能多地练习它的使用。

break语句是C++中一种有效的转移语句，其他的转移语句要么更加危险，要么不如break语句这么有效。

4.4.2 continue语句

continue语句是break语句的一个技术温和的改进。与break类似，它也用在循环结构中。它会跳过在continue语句与循环体尾部之间的循环体的余下部分。

continue语句可以用在while、do-while和for循环中。在while和do-while循环中，它使控制跳转到循环的尾部或者头部以测试循环的条件。在for循环中，continue语句没有跳过增量表达式，它只是跳过循环体的余下部分。

例如，考察程序4-11（没有预读）以及它的修改版本，它们只有在岗哨值还未输入时，才通过修改当前数据来解决问题。

```
while (amount > 0)                          // evaluate current data
  { cout << "Enter amount (negative or zero to end): ";
    cin >> amount;                          // change current data
    if (amount > 0)                         // test validity of data
     { total += amount;                     // process current data
       count++; } }
```

在条件语句中可以不使用块，而是写出这个条件的否定并使用continue语句。

```
while (amount > 0)                          // evaluate current data
  { cout << "Enter amount (negative or zero to end): ";
    cin >> amount;                          // change current data
    if (amount <= 0) continue;              // test validity of data
    total += amount;                        // process current data
    count++; }
```

这并没有大的改善。因为continue语句只是break语句的温和的改进，它总是可以用条件语句来代替。我们不会经常见到使用continue语句能显著改进代码的情形。

4.4.3 goto语句

goto语句是最强大的转移语句。正是由于使用了无限制的goto语句，导致了以下的争论：转移是否是无害的，以及到底要不要禁止它们。实际上，很多现代的程序设计语言禁止无限制地进行转移，C++也一样。

在C++中，goto语句只允许在单个函数的作用域中使用。这意味着goto语句及其目标

（控制流转向的语句）都必须位于同一个函数里。而且，在定义中不允许有任何转移。这相对于传统的转移语句来说有相当大的限制。因此C++的goto语句比旧语言中的goto语句的危害更小。下面是一条C++的goto语句的格式：

```
void foo
{ ...
  goto label1;              // no colon after the label name
  int x;                    // a jump over a definition: syntax error
   ...
  label1: statement;        // a colon after the label name
   ...
  goto label1; }            // no jumps over definitions: OK
```

一个标号是一个标识符；它出现在要转向的那条语句的开头以及goto语句的尾部。程序员为标号命名。不像变量名、函数名和类型名等标识符那样，程序员不必在使用标号之前定义标号。标号标识符有着自己的名字空间。这意味着它们的名字不会与其他的标识符——变量、函数或者类型的名字发生冲突。

冒号应该放在作为转移目标的标号的后面。在goto语句里的标号后面不应该使用冒号，而应该使用分号。

程序4-24给出了事务处理例子的实现，它没有使用循环，只用了条件语句和转移语句。这不是很好吗？

程序4-24 用goto转移语句来处理事务

```
#include <iostream>
using namespace std;
int main ()
{
  double total=0.0, amount; int count=0;        // initialize
start:
  cout << "Enter amount (negative or zero to end): ";
  cin >> amount;                                // enter (change) current data
  if (amount <= 0) goto finish;                 // evaluate current data
  total += amount;                              // process current data
  count++;
  goto start;                                   // go back to the start of loop
finish:
  cout << "\nTotal of " << count << " transactions is " << total << endl;
  return 0;
}
```

有些程序员，特别是那些使用流程图来设计程序的程序员，很喜欢这种程序设计的风格对于这样一个很小的例子来说，是使用转移还是循环，很可能没有什么关系。然而，避免使用goto语句是一个好的建议。只有在使用它们之后会明显改进程序性能时，才使用它们。

4.4.4 return和exit转移

return语句表示终止该return语句所在的函数的执行。如果这个函数是main()函数，该程序就终止。如果这个函数是被main()函数直接或间接调用的其他一些函数，那么就终止该函数，并将控制返回到调用该函数的函数中。

如果函数的返回类型非空，那么函数必须有一条return语句。如果返回类型是void类

型，return语句是可选的。

return语句可能有也可能没有参数。如果函数被定义为void函数，则其return语句必须没有参数。早期的C++从C那里继承了两种形式的main()：一种有int返回类型，而另一种有void返回类型。新的标准C++偏爱第一种形式的main()。但我们还会看到很多有void返回类型的main()函数的C++代码，这些函数没有使用return语句。如果一个带有void类型的main()函数要使用可选的return语句，则其格式如下：

```
void main(void)
{ . . .
  return; }            // no argument, no parentheses
```

当return语句用在一个void类型的函数中时，它必须没有参数和圆括号：return 0是错误的；return()也是错误的。

在前面的那些例子中，使用了返回一个整型值的main()函数。与所有非空返回类型的函数一样，这个main()函数必须有一条return语句，并且该语句必须返回一个整型值（或一个可以转换为整型的数值）。这个main()函数的形式是：

```
int main(void)
{ . . .
  return 0; }          // argument mandatory, parentheses optional
```

实际上，C++编译程序必须（勉强地）接受以下这种形式的main()函数：

```
main(void)             // default return type is integer
{ . . .
  return (0); }        // optional parentheses
```

以前曾经提过，在C++中遗漏了返回类型信息并不意味着没有返回值（void），它意味着返回类型为int。这是从C那里继承下来的特性：一个精简的程序总比一个不精简的程序要好，应该通过提供合适的语言特征鼓励并支持那些想要编写精简程序的程序员这样做。

最近，这种特性正在被另一种观点取代：精简的程序会使维护人员花费更多的时间和精力去理解程序。这意味着对于维护人员来说，精简的程序看起来比不那么精简的程序更加复杂。应该请那些要编写精简程序的程序员多考虑一下程序的可读性和可理解性——特别是从维护人员的角度，因为维护人员可能没有受过很好的训练，或者没有原来的程序员那么有经验。

return语句的返回值可以被调用函数所利用。例如，在前面的例子中调用的cin.get()函数使得我们已讨论的算法可以使用输入的字符值，这意味着在函数get()里，有一些类似于return c的语句；这里，c是某个char类型变量的名字（其名字可以不同）。

当main()函数返回一个值时，操作系统接受该值并判断程序是正常终止还是异常终止。很多平台会忽略程序的返回值。

这并不意味着如果正在编写的函数（包括main()函数）返回一个有类型的值，就可以省略return语句。我们必须记住：如果函数的返回值类型没有被定义为void，函数就应该有一条返回某个适当类型值（表达式）的return语句；在返回表达式两边的圆括号是可选的（但经常用到）。

这里没有限制一个函数可以有多少条return语句。如果一条return语句在函数的中间被执行，那么就不会继续执行函数体的剩下部分。

让我们考察一个原始计算器的简化例子（见程序4-25），该计算器要求用户输入两个操作

数和一个运算符，然后显示运算的结果。为了方便举例，这里使用了没有返回值的main()函数。这个程序的运行结果如图4-22所示。

程序4-25 一个简化了的原始计算器

```
#include <iostream>
using namespace std;
void main(void)

{
 double op1, op2; char ch;
 cout << "Enter operand, operator, another operand: ";
 cin >> op1 >> ch >> op2;
 if (ch == '+')
  cout << "Result is " << op1 + op2 << endl;
 else
  if (ch == '*')
   cout << "Result is " << op1 * op2 << endl;
  else
   if (ch == '-')
    cout << "Result is " << op1 - op2 << endl;
   else
    if (ch == '/')
     if (op2 != 0.0)
      cout << "Result is " << op1 / op2 << endl;
     else
      cout << "Division by zero" << endl;
   else
    cout << "Illegal operator" << endl;
}
```

```
Enter operand, operator, another operand: 22/0
Division by zero
```

图4-22 程序4-25的运行结果（被0除）

我们称这个计算器程序是原始的，是因为它只进行4个算术运算并且不保留计算结果。然而，这个讨论有充分的实际意义。称这个计算器程序是简化的，因为它没有完成一个真正的程序要完成的工作：判断其输入是否有效。对输入有效性的讨论将会偏离我们的主题。

首先讨论一下格式问题。该代码描述了一个深层嵌套的条件语句，每一层嵌套都向右缩进了两个空格。这种格式很好地描述了该程序的结构（嵌套条件语句）。然而，它并没有向维护人员强调这个代码是干什么的。

这段代码从5个操作（加法、乘法、减法、除法、非法操作）中选出一个，但该代码的结构并没有把设计者的信息和意图很好地传达给维护人员。

下面是反映了处理逻辑的条件语句的一个不同版本：

```
if (ch == '+')                          // first case
 cout << "Result is " << op1 + op2 << endl;
else if (ch == '*')                     // second case
 cout << "Result is " << op1 * op2 << endl;
else if (ch == '-')                     // third case
```

```
  cout << "Result is " << op1 - op2 << endl;
else if (ch == '/')                          // fourth case: more complex
  { if (op2 != 0.0)
      cout << "Result is " << op1 / op2 << endl;
    else
      cout << "Division by zero" << endl; }
else                                         // fifth case
  cout << "Illegal operator" << endl;
```

返回语句可以将不同分支的处理变为独立的条件语句，并且它们先后之间不必使用else关键字：

```
if (ch == '+')                               // first case
  { cout << "Result is " << op1 + op2 << endl; return; }
if (ch == '*')                               // second case
  { cout << "Result is " << op1 * op2 << endl; return; }
if (ch == '-')                               // third case
  { cout << "Result is " << op1 - op2 << endl; return; }
if (ch == '/')                               // fourth case: more complex
  { if (op2 != 0.0)
      cout << "Result is " << op1 / op2 << endl;
    else
      cout << "Division by zero" << endl;
    return; }
cout << "Illegal operator" << endl;
```

另外一个流行的终止技术是调用标准库文件stdlib.h中的函数exit()。尽管return语句只终止函数的执行（如果它是main()函数，则程序被终止），而exit()的调用则不管是什么函数调用，都将终止程序。函数exit()可用一个整数参数来调用：根据通常的用法，0表示正常终止，1表示异常终止。通过使用这些值，程序把有关它终止方式的信息传达给了操作系统。

为了适应将来的变化，保护想以这样的方式（调用exit()）和操作系统通信的程序，文件stdlib.h（或者根据新标准的cstdlib）定义了两个符号字面常量：EXIT_SUCCESS和EXIT_FAILURE，建议用它们来取代数值0和1。程序4-26给出了使用这些常量的原始计算器程序。

程序4-26　库函数exit()的调用

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
 double op1, op2; char ch;
 cout << "Enter operand, operator, another operand: ";
 cin >> op1 >> ch >> op2;
 if (ch == '+')                              // first case
  cout << "Result is " << op1 + op2 << endl;
 else if (ch == '*')                         // second case
  cout << "Result is " << op1 * op2 << endl;
 else if (ch == '-')                         // third case
  cout << "Result is " << op1 - op2 << endl;
 else if (ch == '/')                         // fourth case: more complex
  { if (op2 != 0.0)
```

```
        cout << "Result is " << op1 / op2 << endl;
      else
        cout << "Division by zero" << endl; }
  else                                    // fifth case: error
    { cout << "Illegal operator" << endl;
      exit(EXIT_FAILURE); }               // tell them we are bust
  exit(EXIT_SUCCESS);                     // tell them we are OK
}
```

实际上，这些库常量的当前值是0和1。使用这些常量的理由是，可能在某一天因为某个原因，操作系统期望从C++程序中得到一组不同的值；于是，那些使用字面值0和1的程序将陷入麻烦中——因为操作系统将会误解它们。库常量（EXIT_SUCCESS和EXIT_FAILURE）的值可以在库中修改，因此依赖于这些名字的程序总是能够与操作系统正确地通信。这个解释可能有点牵强，但很多程序员都使用这些常量。

4.4.5 switch语句

switch语句是一个在程序中进行多路选择的工具。它基于一个整型表达式的值，提供几条可供选择的执行路径。在圆括号里的表达式跟在关键字switch的后面。该语句的剩下部分置于花括号内的几条分支语句中（开花括号和闭花括号是必须有的）。

每一个分支语句含有关键字case、一个与switch表达式同类型的值、一个冒号以及一个或多个以分号结束的语句。switch语句的闭花括号后面没有分号。下面是switch语句的一般格式：

```
switch(expression) {                      // braces are mandatory
  case ConstantExpr1: statements;         // first branch
  case ConstantExpr2: statements;         // other branches
   . . . . .
  default: statements;                    // default branch
}                                         // semicolon after the closing brace
```

switch表达式只能取以下的类型：char、short、int或long（整数类型）。浮点类型（float、double或long double）是不允许的。程序员定义的类型，如数组、结构或类等，也是不允许的。

例如，在程序4-25和程序4-26的简化的原始计算器中，可以使用带有运算符作为switch表达式的switch语句来实现。程序4-27给出了该计算器的实现。

程序4-27 使用switch的计算器（不理想的程序）

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Enter operand, operator, another operand: ";
  cin >> op1 >> ch >> op2;
  switch(ch) {                                  // mandatory braces
    case '+': cout << "Result is " << op1 + op2 << endl;
```

```
    case '*': cout << "Result is " << op1 * op2 << endl;
    case '-': cout << "Result is " << op1 - op2 << endl;
    case '/': if (op2 != 0.0)
                cout << "Result is " << op1 / op2 << endl;
              else
                cout << "Division by zero" << endl;
  default: cout << "Illegal operator" << endl;
           exit(EXIT_FAILURE); }              // mandatory braces
  exit(EXIT_SUCCESS);                         // next statement
}
```

在case分支中的标号字面值不是变量，它们是编译时的常量（这里是'+'、'*'等等）。在同一个switch语句中，不同的case分支不能够有相同的数值。（如果在不同的switch语句中，它们可以相同。）

在执行期间，switch表达式的值（在本例中是变量ch）自上而下地与case的字面值进行比较。如果表达式与某一个标号匹配，就继续执行跟在标号后面的语句，直到switch语句的结束。

如果没有一个字面值与switch表达式匹配，这不是一个错误。在这种情形下，跟在关键字default后的语句就被执行。default标号是可选的，通常它是switch语句中最后一个标号，但也可以把它放在中间。如果它缺省，并且没有标号与switch表达式的值匹配，则会跳过switch结构中的所有语句并执行下一条语句。注意，switch的case语句不一定都要放在花括号里。花括号不会改变执行的顺序：所有语句都是顺序执行的。

程序4-27的一个执行结果如图4-23所示。我们可以看到C++的switch语句不是一个多分支结构，它是一个多入口结构。如果需要一个多分支结构，则可以通过在switch语句中使用break、goto或return语句终止每一个分支来建立。程序4-28给出了一个更好的switch语句的设计。该程序的运行结果和图4-22中的一样。

程序4-28　使用switch语句的计算器（好一点的程序）

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Enter operand, operator, another operand: ";
  cin >> op1 >> ch >> op2;
  switch(ch) {                                // mandatory braces
    case '+': cout << "Result is " << op1 + op2 << endl;
              break;
    case '*': cout << "Result is " << op1 * op2 << endl;
              break;
    case '-': cout << "Result is " << op1 - op2 << endl;
              break;
    case '/': if (op2 != 0.0)
                cout << "Result is " << op1 / op2 << endl;
              else
                cout << "Division by zero" << endl;
              break;
  default: cout << "Illegal operator" << endl;
```

```
            break; }                          // break is optional here
  exit(EXIT_SUCCESS);                         // next statement
}
```

```
Enter operand, operator, another operand: 22+2
Result is 24
Result is 44
Result is 20
Result is 11
Illegal operator
```

图4-23 程序4-27的运行结果（不正确的程序）

在switch语句中的break语句把控制转移到switch语句闭括号之后的下一条语句。exit()语句像以往那样终止该函数的执行。

goto语句可以用来把控制转移到switch分支之外，但多数情况下，有break语句就已经足够了。有些程序员甚至把break语句放在switch语句的闭括号的前面。这里的break语句是没有用的，但如果有更多的分支加到switch语句中，那么break语句可以防止出错。这不是一个重要的问题，但它是使维护人员的工作变得更加容易的好方法。

如果有多于一个分支要求进行同样的处理，由于switch语句是一个多入口语句而不是一个多分支语句，因此可以用来避免代码重复。例如，考察一个表示用户对应用程序的提示做出反应的变量response。假设当用户输入‘y’或‘Y’时，我们想做某件事；而当用户输入‘n’或‘N’时，我们想做另一件事；并且，如果是其他的反应，就要做其他的事情。处理用户反应的switch语句如下：

```
switch (response) {
  case 'y': case 'Y':
    cout << "Thank you for confirmation\n"; break;
  case 'n': case 'N':
    cout << "Request is canceled\n"; break;
  default: cout << "Incorrect response\n"; }
```

例如，当反应是‘y’时，在case ‘y’: 和case ‘Y’: 之间隐含的空语句被执行，然后执行跟在下一个标号（在本例中是‘Y’）之后的语句。

当然，一系列的条件语句也可以完成同样的操作，但switch语句完成得更好——因为它更易于阅读以及更易于跟踪执行。它是一个很强大的工具。

4.5 小结

关于C++控制流结构还有很多内容可以讨论。C++提供了允许程序员表达复杂算法的所有传统的条件语句和循环语句。C++所特有的是可以把赋值语句放在条件语句和循环结构的逻辑表达式里，再加上C++把任意非0值当做是布尔值true的特征，因此C++程序员可以编写简洁而有说服力的代码。

对初学者来说，要理解这种代码可能有点困难。在学习C++的过程中，很重要的一点是要安排足够的时间来掌握这些特征。C++程序员经常会把注意力集中在学习类和对象上，而忽视了学习这些代表着编写专业性的C++代码的基础。

其他的C++控制结构，包括转移语句和switch语句，对于编写C++代码来说并不像条件语

句和循环语句那样必不可少，我们可以不用它们就能编写出强壮的C++代码。然而，它们是专业技巧方面必不可少的工具。如果没有使用这些语句或者没有正确地使用它们，编写出的代码不可能被认为是专业性的C++代码。大家应该安排足够的时间去学习和练习这些C++语言的元素。

第5章　程序员定义数据类型的聚集

在第4章中，我们学习了使用C++实现算法的一些工具．条件语句、循环语句和转移语句是用来表示如何计算以及以什么样的顺序进行计算的语言结构。在本章中，我们将继续学习如何才能设计定义良好的C++程序，并讨论如何扩展语言的数据类型。

C++允许程序员定义以下的数据集合：数组（同种类的集合）、结构（不同种类的集合）以及派生的数据类型。与以前曾经提过的一样，它们有时被称为程序员定义类型。这是编译程序设计人员的观点而不是程序员的观点。对于编译程序设计人员而言，编译程序的用户就是程序员。对于程序员而言，程序的用户是运行该程序（或使用其结果）的人。因此，这里将程序员在程序中定义的那些类型称为*程序员定义的数据类型*。

我们可以像整型、字符型等内部数据类型一样，定义属于程序员定义数据类型的变量（它们也被称为计算对象，或仅仅称为对象）。

定义这些变量的C++语法是一样的，并且处理变量的规则也一样．实际上，通过程序员定义的数据类型，扩展了不能满足需要的C++数据类型。程序员定义的数据类型也可以进一步用来定义更加复杂的程序员定义的数据类型。在本章中，将主要讨论数组、结构及它们的变体：联合、位域和枚举等。

作为数据和函数集合的C++类，要在我们更加详细地讨论了C++函数之后才能讨论。在第2章中，所介绍的函数对于理解有关函数的基本概念已经足够了，但对于理解类以及用不同方法建立的类还是不够的。

在本章中将要讨论很多内容，并且将更加多样化和复杂化。有人也许想更早地学习有关类的内容，于是想跳过本章中那些不是作为理解类的基础的内容。如果是这样的话，就把精力集中在数组（只是一维的）和结构（但不是层次结构）上。联合以及位域是与类联系较少的程序设计技术。它们并不是不重要，当想拓广程序设计的技能时，可以回到本章继续学习有关的内容。

一般来说，并不需要通过枚举类型去理解类，但C++程序员经常使用枚举类型去定义类组件的大小。在学习第9章时，我们将看到那里使用了一些枚举类型。它们是很直观的，但如果大家需要了解枚举类型的更详细内容，可以回到本章来查阅。

5.1　同种类聚集的数组

数组由一组具有相同数据类型的元素组成。可以把数组看成一组连续的内存单元，这些单元大小相同并表示了同一个类型的元素。我们可以将数组的元素定义为整数类型、浮点类型、字符类型或任何程序员定义的类型——只要在定义数组的源代码中已经定义了这些类型。

5.1.1　作为值向量的数组

我们在第3章中所学习的变量被称为标量或原子变量（简单变量）。

它们只含有一个单值。有时我们需要区别组成一个值的不同构成成分，例如，一个浮点

数的整数部分和小数部分。然而，语言无法实现这一区别，它将这些变量看做是没有构成成分的。因此这些变量被称为标量或原子变量。为了取出一个浮点数的小数部分，我们必须为此而编写一些C++代码。这虽然不是很难（可以利用库函数），但语言中没有提供语言定义的基本方法来实现：

```
fraction = x - floor(x);          // get fractional part of x
```

在这里，x和fraction是双精度浮点数，而floor()是一个在math.h（或cmath）库头文件中定义的函数，它返回不超过其参数大小的最大整数（被转换为double）。然而，该语言把基本类型的值看成是原子。

数组是一个向量：它的状态是用一组值而不是一个单值来描述的。通过使用C++提供的记号（下标运算符），可以立即访问它的每一个元素值。

当每一个元素都要在程序中进行相同的处理时，数组是很有用的。因此数组必须是同种类的：数组的所有元素必须属于同一种类型。于是程序可以访问数组的每一个元素，在每一个元素上完成同样的操作。因此通常利用循环来处理数组的元素。数组的元素属于同一类型，这个事实很重要。它可以防止一个操作适用于数组中的一个元素而不适用于另一个元素时所引起的问题。

数组是数据的有序集合。这意味着数组的每一个元素都有其前一个元素和后一个元素。这里有两个明显的例外：第一个数组元素没有前一个元素，而最后一个元素没有后一个元素。数组有一个名字，但单个数组元素没有单独的名字。程序存取它们是通过附加下标的数组名来实现的，下标表示了元素在顺序集合中的位置。

数组的长度是有限的。数组中元素的数目在编译时必须是已知的，并且在程序执行的过程中不能改变。程序员必须决定在数组中将存放多少个元素，在编写程序时给出约定并遵守这一约定。

这是一个严格的限制。如果程序员为数组分配了太多的空间，过多的空间将造成浪费，并且程序可能没有足够的内存去完成其他工作。如果程序员没有分配足够的内存，程序在执行时就会破坏内存，应用程序就可能会崩溃或产生不正确的结果。如果程序员想改变数组的大小，就只能通过编辑程序的源代码、重新编译并且重新连接。对一个小程序来说，这是简单的；但对于一个复杂的程序或者一个已分发给几千个用户的程序来说，这种修改是非常困难的

有时候，数组的大小是精确可知的。例如，表示一个星期每一天的工作时间就应有7个元素（除非以后规定每个星期要加上一两天）。对于表示一个月的天数的数组，也有同样的情况（除非一年中的月数改变了）。对于表示国际象棋棋盘的数组，也是一样。然而，在大多数情况下，我们需要去寻找一个“合理的”折中：分配的元素数目比我们估计所需要的多，但不会太多（双倍于该数）。如果代码支持这个约定，并且当发生溢出时能够采取一个“合理的”行动，该折中就是“合理的”。对于某些人而言，“合理的”行动可能意味着程序终止；对于其他人而言，它可能意味着通知用户终止输入。

有时，一个元素在数组中的位置与应用有关。例如，在给定的某天管理医院病房的医生的名字是和一个星期的某一天相关联的。当输入数据时，它们并不是按先后次序输入的。在某些数组元素中可能没有有效的数据。当使用这样的数组时，我们必须让程序能够区分已存放有效数据的元素和没有存放有效数据的元素，这样的数组被称为稀疏数组。

大多数数组是连续的数组。第一个存入数组的数据项被存放在第一个元素中，下一个数据项被存放在下一个元素中。在这里，我们也需要区分含有有效数据的元素和不含有效数据的元素。连续存储的优点是不需要标记每一个数组元素是有效的还是未用的，在某一个指定位置之前的所有数组元素都是已用的，而在那个位置之后的所有数组元素都是未用的。

对连续存储的数组有两种实现方法。一种是对已插入到数组的有效值进行计数，这样处理该数组有效元素的循环可以利用这个计数去终止循环。另一种实现连续数组的方法是使用一个特殊的值，把它插在数组的最后一个有效元素之后，当循环遇到这个特殊数值时，对有效的数组元素进行处理的循环就会停止。这个特殊的值称为岗哨值（它类似于在第4章中用来决定输入结束的标记），它的取值应该与数组元素所取的有效值不同。

5.1.2 C++数组的定义

和所有C++变量一样，数组变量必须在使用之前定义。数组定义把数组名、数组元素的类型以及元素个数联系在一起。同样，数组定义的作用是为该数组在执行时分配内存空间，且数组定义以分号结束。可以用单独一行定义一个数组，或者可以把几个定义写在同一行中，例如：

```
int hours[7]; char grade[35]; double amount[20];
```

这一行定义了3个数组：有7个整数元素的数组hours[]、有35个字符元素的数组grade[]以及有20个双精度浮点数元素的数组amount[]。请注意这里写在数组名字后的空方括号，这个符号表示该变量是一个有若干个值的向量，而不是一个单值的标量。

对于不同类型的数组来说，像前面的例子一样，每一个数组都必须分别地定义，并以分号来结束其定义。对于属于同一个类型的数组，可以用逗号分隔若干个数组的定义（以及用分号来结束最后一个定义）。实际上，如果类型相同，还可以把数组和标量的定义写在一起，例如：

```
int category[7], i, num, scores[35], n;
```

有些程序员用复数形式来对数组命名，因为当把数组作为参数传递给函数时，例如，sum(scores)，它表示函数得到的是一组成绩而不是单独的一个成绩。其他一些程序员则使用单数形式来对数组命名。比如，当通过下标引用数组的一个单独元素时，例如category[i]，它表示要操作的是一个类别而不是一组类别。使用单数还是复数对数组命名，不是一个十分重要的问题。

尽管数组的大小必须在编译时已知，但它还不具有一个字面值。它可以是一个预定义的符号文字、整数常量或任意复杂度的整数表达式。惟一要求的是，必须在编译时而不是在运行时确定表达式的值。例如：

```
#define MAX_RATES 35                    // array size as a #defined value
int const NUM_ITEMS = 10;               // array size as a constant
int rates [MAX_RATES]; double amount[2*NUM_ITEMS];
```

数组可以像其他C++变量一样在定义时初始化，程序员可以像初始化标量那样提供初始化值。这些初始化值可以写在花括号内并以逗号隔开。由于逗号是分隔符而不是终止符，因此在闭括号之前的那个初始化值之后没有逗号。

```
int hours[7] = { 8, 8, 12, 8, 4, 0, 0 };        // 7 values
```

```
int side[5] = { 40,35,41 } ;                    // other array elements are 0's
char option[2] = { 'Y', 'N', 'y', 'n' };        // syntax error
int week[52] = { , , 40, 48 };                  // syntax error
```

第一个初始值初始化数组的第一个元素，第二个初始值则初始化第二个元素，以次类推。初始值的类型应该与数组类型一样，如果类型不同，则应该允许两个类型的值之间进行转换。这些转换和第3章中所讨论过的表达式中的混合数值类型的转换是一样的。（例如，用整数初始值初始化double类型的数组元素是允许的。）

在这些例子中，为数组hour[]的每一个元素都提供了值。也可以提供少于元素个数的初始值，就像数组side[]那样：从第一个元素开始初始化，直到所有的初始值用完为止。那些剩下的没有初始值的元素则初始化为0。不能提供多于数组元素个数的初始值，比如像数组option[]那样。也不允许通过使用逗号来跳过某些元素，比如像数组week[]那样。尽管作业控制语言（JCL）允许这种语法，但C++不是JCL。

类似于标量变量，在某个文件中定义的数组变量可能会在另一个文件所实现的算法中使用。为了使之成为可能，另一个文件必须用同样的名字声明该数组变量。数组定义和声明之间的主要差别是声明没有确定数组的大小。数组声明并没有为数组分配内存。（这是数组定义的任务。）虽然C++的声明和定义相类似，但程序员应该能够把它们区分开来。

例如，某个文件可能需要数组hours[]的元素值，或者它可能要计算这些元素的值。在这个文件里，数组hours[]将会这样声明：

```
extern int hours[];             // declaration: no memory allocated
```

为了使这个声明合法，数组hours[]应该是一个全局变量，并且它的初始定义应该置于任何函数之外。

类似于标量变量的声明，数组声明的作用是在内存中建立该数组的地址。现在这个文件中的代码可以存取数组hours[]的元素，就像该数组是定义在这个文件中一样。由于数组声明（和其他所有声明一样）没有分配内存，它们不支持初始化操作。

但是，C++允许程序员用声明的语法来定义数组。当数组的大小可由初始化值确定而不是由一个显式的编译时常量确定时，程序员就可以这样做，例如：

```
double rates[] = { 1.0, 1.2, 1.4 };             // three elements
```

在这里，尽管是数组rates[]的声明形式，但为数组的3个元素分配了空间并作了初始化。这个定义等价于下面的定义。

```
double rates[3] = { 1.0, 1.2, 1.4 };            // explicit count
```

第一种定义的优点是节省了输入数组大小的几次按键。而另一方面，第一种定义放弃了为数组大小定义一个常量的机会，而且这样的常量在处理数组的算法中经常用到。

解决这个问题的一个方法是使用第3章中见过的sizeof运算符来计算数组元素的个数。用一个元素的大小去除数组的大小，便可以得到数组元素的个数。

```
int num = sizeof(rates) / sizeof(double);
```

请注意对C++数组的讨论次序，它类似于对其他数据定义机制的讨论。我们首先讨论所要介绍的新的C++机制的含义（包括第3章中的变量，本章中的数组，然后是结构、类、复合类

和派生类），接着是定义（和声明，如果有必要的话）的语法，然后我们讨论初始化问题。这个讨论的顺序不是偶然的。初始化在C++中非常重要，我们将会学习与C++的每一种内存使用有关的初始化方法。

5.1.3 数组上的操作

讨论了初始化问题之后就应接着讨论数组的操作。我们可以对数组进行什么操作呢？在这方面C++的能力是很有限的。不能把一个数组变量赋值给另一个数组，而且不能够比较两个数组，也不能将两个数组相加、相乘等等。对于一个数组，惟一能做的就是把它当做参数传递给一个函数。因此当需要将一个数组赋值给另一个数组、或者要比较两个数组时，要编写代码或者使用库函数来实现——如果它们是可利用的。

所有的操作都能针对单个数组元素进行。当我们将一个数组拷贝到另一个数组时，是逐个地拷贝每一个数组元素的。当我们比较数组时，要逐个地比较相应的数组元素。在这些操作中，我们通过使用下标运算符去引用每个数组元素。

例如，side[2]代表数组side在下标2上的元素。不论如何，side[2]都是一个普通的标量整型变量。由于数组side[]是一个整型数组，所有可以对整数进行的操作都可以应用在side[2]上。它只是名字的形式与以往的不同：这里使用的是数组名字加上下标以及下标运算符，而不是一般整型变量所用的标识符。

```
side[2] = 40;                          // use as lvalue
num = side[2] * 2;                     // use as rvalue
```

在第一行中，side[2]获得值40并存放在它对应的地址上。在第二行，存放在side[2]地址上的值被乘以2，并将结果存放在变量num（它必须是数值类型的）中。正如我们所见，单个数组元素并没有单独的名字。它们的名字由数组名与下标值组成。

C++的数组元素记号是常见的。不寻常的是，C++把方括号看做是运算符而不是一种符号。如果查阅第3章中的表3-1，我们将看到这个运算符是高优先级的，它处于C++运算符表的顶部和其他运算符一样，下标运算符也有操作数。它们是什么呢？它们是数组名字和下标值。该运算符被用于名字side和值2上，运算的结果是side[2]，它表示了数组元素的名字。

这听起来很抽象和远离实际。它是一个运算符，还是一个特殊符号又会有什么差别呢？目前它们没有什么差别。不久，我们将在一些有趣的环境中使用这个运算符。

下标并非一定是一个字面值甚至或者是编译时的值，任何运行时的数值表达式都可以被作为下标来使用。如果表达式是浮点数、字符、短整数或长整数值，它将会转换为整数。例如，在这里运行时调用了函数foo()，其返回值被用来计算下标。

```
side[3*foo()] = 40;                    // is this legal?
```

如果要使这种写法合法的话，函数foo()必须已经定义并且其返回值（下标值）应该在合法的下标值范围之内。如果只是数组的一部分元素被赋值，该索引就必须是这些元素中的某个下标。如果所有的数组元素都被赋值，该下标就必须介于第一个元素和最后一个元素之间。在这个范围之外的下标值引用了不在该数组的内存单元，因此不应该用来引用数组元素。

5.1.4 下标正确性的检查

请注意，程序员不能随意地为一个数组选择其下标值范围；对于所有的C++数组来说，它

是固定的。这令人很不舒服，因为我们经常想把某些值赋给下标。例如，我们可能有一个存储着从1997年到2006年的税收数据的数组revenue[]；把从1997年到2006年作为数组下标的范围可能会更方便。其他语言允许程序员选择下标范围，但C++不允许。在C++里，下标的范围是固定的。而且，它必须从0开始。

也就是说，任何C++数组的第一个元素的下标是0而不是1。这一点很重要。

例如，如果数组side[]有5个元素，则其合法的数组元素是side[0]、side[1]、side[2]、side[3]和side[4]。注意，side[5]并不是这个数组的合法元素。

如果犯了错误而引用side[-1]、side[6]或者是side[5]，会怎么样呢？编译程序会指出有错误吗？不。下标值可以是运行时的值，在编译时是不确定的，而且编译程序不会检查下标的正确性。

C++以尊重程序员的姿态跳过这个正确性的检查。如果在代码中写了side[-1]，很明显是想用它来表示某些事，而再去猜测程序的意图并指出有错并不是编译程序的任务。

有没有运行时的检查呢？在C++中没有，因为在运行时验证下标的正确性将影响程序性能，而这正是C++所极力避免的。如果想在运行时检查下标的正确性，就只能自己动手。

当然，这种语言的潜在假设是程序员每一刻都知道他或她正在干什么，并不需要从编译程序或运行时的系统那里得到什么帮助。毫无疑问，这个假设是完全没有根据的，并且下标处理中的错误是C++程序员一个常见的出错原因。

这样做（从C继承的）的原因是数组名被用作数组的第一个元素的地址。第一个元素的位移就是0。第二个元素的位移是一个元素的长度（依赖于它的类型）。第三个元素的位移是两个元素的长度。编译程序知道元素的大小，而且用位移来计算元素的地址比用元素在数组中的序号来计算会更加简单。

当下标值不正确时，编译程序仍然用这个下标作为位移去计算元素在内存中的地址，因此程序会破坏它的内存。然而，如果这个地址没有用来做某些有用的事，就可能侥幸地避免这个问题的发生。

警告　在C++中没有编译时对下标的正确性检查。在C++中没有运行时对下标的正确性检查。计算机的内存可能会被用户程序破坏。请注意！

让我们考察一下在处理下标时出错的结果。程序5-1给出了一个程序，它正确地给一个多边形的各条边赋值，但不能正确地打印它们：因为第一个值的下标是1，最后一个值的下标是5。程序的输出如图5-1所示。

程序5-1　对数组的错误扫描

```
#include <iostream>                          // or #include <iostream.h>
using namespace std;

int main()
{
  int size[5] = { 39, 40, 41, 42, 43 };
  for (int i = 1; i <= 5; i++)               // bad start, bad end
    cout << " " << size[i]; cout << endl;
  return 0;
}
```

在这个例子中，输出的检查指出代码中有错。但是，如果程序员总是犯这样的错误，那

么显示输出的检查就不会引起程序员的警觉。程序5-2是一个不正确地对多边形的各边赋值和不正确地打印它们的程序。该程序没有使用属于该数组的side[0]单元，却使用了不属于该数组的内存单元side[5]。如图5-2所示，其输出是正确的，尽管该程序破坏了它所引用的side[5]的内存单元。

```
40  41  42  43  13540
```

图5-1 输出显示了代码中的错误

程序5-2 错误被正确的输出所隐藏

```
#include <iostream>                          // or #include <iostream.h>
using namespace std;

int main()
{
  int size[5];
  size[1]=39; size[2]=40; size[3]=41; size[4]=42; size[5]=43;
  for (int i = 1; i <= 5; i++)               // bad start, bad end
    cout << " " << size[i]; cout << endl;
  return 0;
}
```

```
39  40  41  42  43
```

图5-2 正确的输出隐藏了数组处理中的错误

讹用内存的危险有多大呢？如果这个代码破坏的内存没有被任何有用的程序占用（许多机器中有很多内存没有分配给有用的程序），这就没有问题。如果讹用的内存被某个程序所使用，那错误就很难找了。如程序5-2所示，这里很难发现程序是不正确的，也很难确定该从哪里开始寻找错误。程序5-3扩充了这个例子。与图5-3一样，a[0]的值是不正确的：它从11变为43，尽管没有对a[0]进行第二次赋值。在一个实际处理中怀疑对数组side[]的处理会改变数组a[]的值是不大可能的。在不同的计算机上，这个程序可能以不同的方式去讹用内存。不管它做了什么，这个看起来没有什么错的小程序却是不正确的。

程序5-3 一个地方的错误讹用了另一个地方的内存

```
#include <iostream.h>
void main()
{ int a[3]; int size[5];
  a[1]=11; a[2]=12; a[3]=13;                 // a victim of corruption
  size[1]=39; size[2]=40; size[3]=41; size[4]=42; size[5]=43;
    for (int i = 1; i <= 5; i++)             // bad start, bad end
      cout << " " << size[i];
    cout << endl;
```

```
for (i = 0; i < 3; i++)                    // correct start, end
  cout << " " << a[i];
cout << endl; }
```

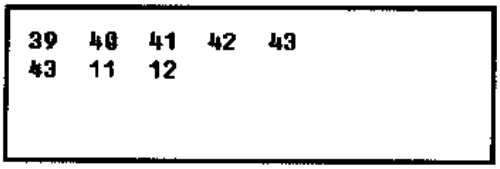


图5-3　数组a[]被数组side[]的操作破坏了

处理数组元素的正确循环应该使下标从0而不是从1开始。循环应该以一个小于数组大小的值结束。如果数组大小是5，测试的正确形式是i<5；如果数组大小是3，测试的正确形式是i<3。通常，如果有效的数组元素的数目是变量NUM，循环测试的正确形式就是i<NUM。在程序5-3中，对数组a[]的循环设计是正确的。注意下标i是在第一次循环中定义的，而不是在程序的开头。它的名字直到该函数结束之前都是可知的。因此，第二个循环没有定义这个变量但却还可以使用它，就像它是在函数的开头定义一样。作者的编译程序（微软Visual C++ 6.0版）没有正确地实现标准的C++：下标变量i的作用域只能在第一次循环中，而不能在整个函数中。

C++程序员必须总是要注意下标的正确性。当对数组的所有元素进行处理时，应该从下标0开始循环，应该用比元素数目少1的下标来结束循环。这是一个很简单的规则，既不难记住，也不是很难使用。大多数时候我们都可以做得很好，但有时，程序员在存取数组元素时容易犯错误，而为这些错误所付出的代价是非常高的，特别是在维护程序的时候。如果将在处理数组下标出错上所花费的所有时间、努力和所受的挫折都加起来，其结果将是令人吃惊的。因此C++程序员一定要注意下标的正确性。

提示　*在数组上的循环应该从下标0开始。当下标值小于有效的数组元素的个数时，可以继续进行循环。*

5.1.5　多维数组

C++支持多维数组。从理论上说，对数组的维数没有限制。当定义一个多维数组时，要定义数组元素的类型、数组名字以及位于方括号中的第一维的元素个数、第二维的元素个数等等。例如，一个2行3列的整数类型的二维数组可定义为：

```
int m[2][3];                // 2 rows of arrays, 3 elements each
```

多维数组可以使用类似于一维数组初始化的语法进行初始化：初始值使用逗号分隔符分隔，并列在一个块中。

```
int m[2][3] = { 10, 20, 30, 40, 50, 60 };
```

在这里，前面3个值对应于矩阵的第一行，后3个值对应于第二行。对于更大的数组，可以通过使用作用域花括号指出属于同一行的一组值。各行的初始值用逗号分隔开。这有助于维护人员更容易识别每一行上的数据。

```
int m[2][3] = { { 10, 20, 30 }, { 40, 50, 60 } };
```

类似于一维数组，可以对每一行给出比应有的元素个数少的初始值。剩下的元素将被赋值为0。例如：

```
int m[2][3] = { { 10, 20 }, {30, 40 } };
```

这等价于下面的显式定义，其中前3个值对应于第一行，后3个值对应于第二行。

```
int m[2][3] = { 10, 20, 0, 30, 40, 0 };
```

类似于一维数组，不能给出比一行应有的元素个数还要多的初始值。否则将是一个语法错误。

```
int m[2][3] = { { 10,20,30,40 }, { 50,60 } };        // error
```

通过给出初始值来定义数组大小的方法也可以用于多维数组，但只是部分地起作用。我们可以省略行数，但必须给出列数，编译程序将会计算出初始值的个数并且计算出行的数目。

```
int m[][3] = { { 10, 20, 30 }, { 40, 50, 60 } };
```

不管是否给出行的数目，我们不能省略列的数目，否则将是一个语法错误。

```
int m[2][] = { { 10,20,30 }, { 40,50,60 } };         // error
```

编译程序可以计算出每行的列数并得到矩阵的结构，但却没有这样做。不管怎样，显式地给出数组的维数可能会更好。

存取多维数组的元素需要有几个下标，一维一个。类似于一维数组，每一个下标代表该元素的位移，因此它们从0开始并在每一维里应以比该维的元素个数小1的下标来结束循环。例如，在矩阵m[][]里第二行的第一个元素表示为m[1][0]，它既可以作为右值（作为表达式中的一个操作数），又可以作为左值（作为赋值运算的目标）。

在遍历一个多维数组的元素时，应该使用嵌套循环。在多维数组的嵌套循环中，内层的循环首先执行，内层循环结束后，就进入外层循环的下一次循环继续执行。

程序5-4给出了一个以行为主序的方式显示矩阵m[][]的每一个元素的嵌套循环，内层循环为每一个外层循环的下标值i（从0变为1）修改下标值j，使之从0变为2。该程序的输出如图5-4所示。

程序5-4 二维数组操作的例子

```
#include <iostream.h>
void main()
{ const int ROWS = 2, COLS = 3;
  int m[ROWS][COLS] = { { 10, 20, 30 }, { 40, 50, 60 } };
  for (int i=0; i < ROWS; i++)                    // done once for each row
    { for (int j=0; j<COLS; j++)                  // done for each index i
        cout << " " << m[i][j];
      cout << endl; }                             // end of row: once for each index i
}
```

```
10  20  30
40  50  60
```

图5-4 二维数组以行为主序的显示

从其他语言转到使用C++的那些程序员会觉得多维数组的形式有点难理解（或只是新颖）。他们有时只使用一组括号并用逗号隔开下标。例如，可能写为m[1,0]，而不是m[1][0]。不幸的是，编译程序并不会指出有错。编译程序只是将这个逗号表达式的值作为下标值，使得程序运行时出错。图5-5给出了将程序5-4中m[i][j]误写为m[i,j]后的程序的输出结果。

```
0x34C4  0x34CA  0x34D0
0x34C4  0x34CA  0x34D0
```

图5-5　将程序5-4中的m[i][j]误写为m[i,j]后的输出结果

造成这个意外的原因有两个，它们都是从C那里继承下来的。一个原因是逗号是一个C++运算符。当编译程序对用逗号分隔的下标表达式[i,j]求值时，它首先计算i（或1）的值，然后发现那个逗号，于是丢掉i的值并计算下个表达式j（比如0）的值。然后编译程序将该值作为下标。于是将m[i,j]理解为m[j]。第二个原因是只有一个下标的m[j]是位移为j的某一行的合法记号，因为多维数组不要求给出所有维。

注意　当引用一个二维数组的某个元素时，必须使用两组方括号：a[i][j]。不要使用逗号：因为a[i,j]会导致不必要的麻烦。

多维数组在C++中有语法支持只是为了程序员的方便。在机器里，它们是用一维数组来实现的。有些程序员宁愿使用有ROWS*COLS个元素的一维数组，并将第i*COLS+j个元素作为第i行第j列的元素。（不要忘记：下标从0开始并以ROWS*COLS-1结束）。程序5-5给出了与程序5-4一样功能的程序，其中的数组被当做一维数组进行处理。这个程序的输出和图5-4中的一样。

程序5-5　用一维数组实现一个矩阵

```
#include <iostream>
using namespace std;

int main()
{
 const int ROWS = 2, COLS = 3;
 int m[ROWS * COLS] = { 10, 20, 30, 40, 50, 60 };   // same size
 for (int i=0; i < ROWS; i++)
  { for (int j=0; j < COLS; j++)
      cout << " " << m[i*COLS + j];                 // do it hard way
    cout << endl; }                                 // end of row: done once for each i
 return 0;
}
```

哪种处理下标的方式更好呢？这取决于应用的需求和个人的喜好，因此很难说哪一个更好。通常，使用数组描述时，是用一维的还是多维的数组，是使用一个下标还是几个下标，都是不重要的。我们只需做好处理下标的准备。

5.1.6　字符数组的定义

字符数组的重要性基于以下的事实：在C++中文本是用字符数组表示的（它们常被称为字

符串），本章中有关数组（一维或多维的）的所有讨论也适用于字符数组。

不论对数组进行什么处理——打印，保存到文件中，拷贝到另一个数组，与另一个数组比较等等，程序员都必须知道数组在什么地方结束。通常，因为数组的大小应该比实际存在数组里的元素要多，因此在数组中元素的实际数目通常比数组的大小要小。

对这个问题有两个处理方法：一个方法是记录数组中实际元素的数目，另一个方法是在数组的结尾使用一个岗哨值。对于非字符数组，两个方法都可用。对于字符数组，C++使用的是第二个方法。C++用数字0作为字符数组的岗哨值，因为0是一个不同于任何合法字符编码的特殊编码。（它经常被称为0终止符或终止符。）

为了将这个编码与字符'0'（ASCII码，十进制数48、十六进制数30）相区别，岗哨值字符通常表示为转义字符'\0'（十进制数0，十六进制数0x0）。

当字符数组或字符串作为参数传递给任何一个C++库函数时，这些函数要求该字符串要以这个标记字符作为结尾。不管函数何时产生字符数组，都会把这个标记添加到数组中字符串的结尾，以便该数组可以被其他库函数使用。

```
char t[4] = { 'H','i','!','\0' };          // four array elements
cout << t << endl;                         // It displays "Hi!"
```

这里，数组t[]使用一维数组的标准语法来初始化，然后作为一个参数传递给运算符函数<<。这个函数连续地逐个打印字符串的字符，直到发现编码0时它就停止打印。

转义字符的形式只有在这个值是一个字符串的一部分时才是必不可少的。在很多情况下，可以使用数字0。例如，以下使用编码0来代替转义字符（'\0'）是可以的。有些程序员喜欢使用字符记号。

```
char t[4] = { 'H', 'i', '!', 0 };          // some prefer '\0'
```

这种记号对于初始化较长的字符数组是不方便的。意识到这种实际需要，C++允许这样做初始化：我们可以用字符串作为一组字符的初始值。编译程序能理解其中的意思，把每一个字符放到数组的对应位置上，并且在结尾加上0终止符。

```
char t[4]="Hi!";                           // t[0] is 'H', t[1] is 'i', etc.
char u[]="Today is a nice day.";           // 21 characters with terminating 0
```

字符串"Hi!"有4个字符，第4个是编码0。字符串"Today is a nice day."包括编码0在内共有21个字符，因此数组u[]有21个元素，而不是20个。岗哨字符需要一个额外的数组元素空间来存放。如果不为这个额外的元素提供存储空间可能会引起问题。例如，这是一个语法错误：

```
char v[3]="Hi!";           // Four initial values for 3 elements
```

用多于初始化字符所占的空间去定义字符数组是可以的。定义一个字符数组而其内容没有定义也是可以的。

```
char last[30]="Jones", first[30];       // space is available
```

当然，存取字符串元素的情形类似于普通的数组。每一个数组元素属于char类型，第一个元素的下标是0。

```
t[0] = 'N'; t[1] = 'o';          // t[] contains "No!" now, not "Hi!"
```

当处理字符和字符串时，要区分单引号和双引号的使用。例如，上面的'0'是一个字符，

但"0"是一个字符串，它包含两个字符：字符'0'和字符'\0'。

5.1.7　字符数组上的操作

单独的字符可以相互赋值或与其他字符相互比较，可以进行移位和添加等操作。这些操作没有一个可用于字符串（字符数组）。幸运的是，C++库提供了大量的对字符数组进行操作的函数。当作为函数的参数使用时，数组名可以不带下标。

函数strcpy()为字符数组实现赋值运算。它把两个数组作为参数，并把第二个参数的元素复制到第一个参数的对应元素中，复制操作不断进行，直到在第二个参数中发现编码0为止。0终止符也同样被复制，使目标数组成为一个可以用作其他函数参数的合式的数组。

```
strcpy(u,t);                    // Now u[] contains "No!", too
```

因为没有函数会去检查在标记之后字符串的内容，所以没有必要清除串的剩下部分。在这个函数调用之前，串u[]的内容是"Today is a good day.\0"，其中“\0”是标记字符。在该函数调用之后，它的内容变成'No!\0y is a good day.\0"，但在第一个“\0”之后的内容已不再重要。（这里，转义字符的使用是必不可少的）。无论如何，这个串的内容都是“No!”。

把一个字面字符串作为参数传递给一个要求以字符数组作为参数的函数是没有问题的，因为每一个字面字符串的结尾都有0终止符。惟一的要求是该函数不能改变数组的状态，因为字面字符串是常量因此其内容不能被函数改变。

```
strcpy(t,"Yes");                // Now t[] contains "Yes" plus zero
strcpy("Yes",t);                // No, you cannot do that: syntax error
```

函数strcat()对字符数组实现+=操作。它把两个字符数组作为参数，并将第二个参数复制到第一个参数中。不像strcpy()那样，它不是用新的内容代替第一个参数的内容，而是把它们添加到现有内容之后。其结果是两个串的串接。

```
strcat(u," means No!");         // u[] contains "No! means No!"
```

这里在串接之前是在终止符的位置上开始添加字符，终止符被放在添加的诸字符之后，使得数组u[]的内容变为“No! means No!\0d day.\0”由于没有C++函数会检查终止符之后的内容，因此无论如何，该串的内容都是“No! means No!”。

函数strcmp()实现两个字符数组参数之间的相互比较操作。它逐个地比较相应位置上的字符，直到在某一个位置上遇到两个不同的字符或者到达岗哨值为止。如果函数发现了两个不同的字符，就会比较它们的字典顺序，也就是在ASCII码表中哪一个的位置在前面。如果是升序的，即第一个参数的字符先于第二个参数里的字符，那么strcmp()返回-1；如果是降序的，即第二个参数里的字符先于第一个参数里的字符，strcmp()就返回1。如果函数在同一个位置上同时到达标记，就返回0，表示两个串是一样的。

例如，strcmp("Hi", "Hello")返回1；因为它们是降序的。另一方面，strcmp("Handler","Hello")返回-1，而strcmp("Hell","Hello")也一样，因为这里用第一个串中的岗哨值与第二个串的字符‘o’进行比较。由于在ASCII表中小写字母跟在大写字母之后，因此strcmp("hello","Hello")返回1，以此为条件的条件语句的true分支将被执行。

```
if (strcmp("hello","Hello")) cout << "Not ordered\n";
```

注意 所有C++库函数在发现终止符0时，就会停止字符串的处理。在终止符0之后的符号对库函数来说都是不可用的。使用strlen()可以求出在终止符0之前的字符个数

另一个有用的库函数是strlen()，它接受一个字符数组作为参数并返回字符串中位于0终止符之前的字符个数。例如，strlen("Hel o")返回5，strlen(t)返回3（t含有"Hi!"）。所有库函数一旦发现岗哨字符就会停止检查字符串的内容。

5.1.8 字符串函数和内存讹用

没有一个函数会检查是否有足够的空间进行所需的操作，原因是C++函数不会知道它接收的参数数组中元素的总数，不管它是字符数组还是其他类型的数组。但真正的原因是这种检查会影响程序性能，因此C++程序员总是要考虑是否有可用的空间并确保有足够的空间可用

另外，如果两个数组在内存里重叠，这些函数就会给出"不确定的结果"。这意味着其结果可能是正确的或不正确的，但任一种情况都无法确定。

程序5-6给出了一个不顾后果地处理可利用空间的函数。还有什么处理比输入两个数据项然后回应它们更简单的呢？该程序把数组first[]和last[]传递给抽取函数>>——它用输入字符来填充数组并为它们添加0终止符。

程序5-6 数组溢出的一个简单例子

```
#include <iostream>
using namespace std;
int main()
{
 char first[6], last[6];                    // are not these arrays too short?
 cout << "Enter first name: ";              // I enter "John\0" (5 symbols)
 cin >> first;                              // no protection against overflow
 cout << "Enter last name: ";               // I enter "Johnson\0" (8 symbols)
 cin >> last;                               // no protection against overflow
 cout << first << " " << last << endl;      // just to check results
 return 0;
}
```

该程序的输出如图5-6所示。当数据足够短时，没有问题。当数据较长时，问题（在这个无足轻重的例子中）就会变得明显。有人可能认为6个字符对于姓名是不够用的，而实际上，只有5个字符可以使用，因为第6个被终止符占用了。大家是否认为20个字符够用了呢？对于作者一个朋友的名字Galina Belosel skaya-Belozerskaya，就会出现溢出的问题，因为包括空格和0终止符在内，她的名字共含有33个字符。

```
Enter first name: John
Enter last name: Johnson
n Johnson
```

图5-6 输入数组的溢出无警告地破坏了其他数据

如果从用户的键盘输入又会怎样呢？用户可能会看到并改正这种明显的错误，但这时内存可能会无警告地被破坏。

这里，数组first[]含有字符串"John\0"（最后两个符号'\0'代表一个内容为0的内存单元）。数组last[]含有"Johnson\0"（包括0终止符在内共有8个字符）。然而，数组last[]的空间只能存放6个字符。"n\0"这两个字符去了哪里了呢？在作者的计算机的内存中，数组first[]实际上是跟在数组last[]之后的。为何会这样并不重要，我们可以确定的是这两个字符已经去了某个地方。在输入姓后，数组first[]含有这两个字符"n\0"以及从"John\0"留下的内容，也就是"n\0hn\0"。当作者用cout打印数组first[]时，如果发现第一个"\0"，就在打印了'n'之后立即停止。

这一点很有趣，但也是相当危险的。在不同的计算机上可能会有一些差别。有些编译程序以4个字节为单位来分配空间，因此该数组实际上每个元素含有8个字符而不是6个，我们必须使用一个更长的名字才能观察到内存的破坏，而有些编译程序没有把数组first[]放在数组last[]之后。不论何种情况，重要的是字符串处理有破坏内存的倾向。

动态内存管理是解决这个问题的一个好方法，但我们还没有所需的工具。另一个实际的解决方法是限制可以存放到数组里的字符个数。这可以通过使用输入函数get()实现，该函数允许程序员指定输入字符个数的上界。

```
cin.get(first,6);          // read up to 5 characters + null
```

如果函数get()在字符个数到达上界值减1之前发现换行字符，它就停止输入，而换行字符'\n'留在输入缓冲区中作为下一次输入的第一个字符。另一个空终止符就会被添加到数组的末尾，而不会有任何问题。

如果用户不断地输入而不按Enter键，当输入的字符个数到达上界值减1时，输入就终止。这时也会加上空终止符使得该字符串是合式的。当用户最终按下Enter键时，那些多余的输入字符被保留在输入缓冲区里，并以换行符为结束。它们将被下一条输入语句（如果有的话）读取。

使用函数get()会引起两个问题。假设名字只含3个字符（比如，"Amy"），然后输入姓氏：

```
cin.get(last,6);           // it stops when it finds new line
```

我们会看到输入缓冲区中的第一个字符就是前一次读取"Amy"的get()调用所剩下的换行符，get()的这次调用读取了换行符后就终止了。结果，空字符串被读入到数组last[]中。随后不论用户输入什么，都不会进入数组last[]中而只会保留在输入缓冲区中。如果第一次输入得太长（"Vladimir"），那么只有5个字符和0终止符被放入数组first[]中。输入的剩余部分（"mir"）将会留在输入缓冲区中等候下一次的输入请求。不论用户输入什么姓氏，都不会存放在数组last[]中。程序将从输入缓冲区中读取字符（"mir"），并在遇到换行符（它被留在那里）处停止。

可见单独用函数get()不能解决问题，还需要用函数ignore()，该函数读取输入字符并将它们去掉。它要求用户指定要去掉的字符个数的上界以及停止去掉字符的定界字符。（在这里，就是换行符。）

程序5-7给出了对输入数据溢出问题的解决方法。它同时也给出了另一个与复制和串接有关的问题。该程序用姓氏、逗号、空格和名字表示顾客的姓名。如果被复制到数组name[]

的字符个数超过数组的大小，那些字符仍然被复制到一个刚好邻近该数组的内存空间中。图5-7给出了执行的结果。尽管数组name[]含有“正确的”数据，数组last[]被无警告地破坏了，虽然程序没有显式地改变其内容。

程序5-7 在串接中导致数组溢出的一个例子

```
#include <iostream>                                  // or #include <iostream.h>
#include <cstring>                                   // or #include <string.h>
using namespace std;
int main()
{
 char first[6], last[6]; char name[10];              // name = last, first
 cout << "Enter first name: ";
 cin.get(first,6); cin.ignore(2000,'\n');            // has to remove CR
 cout << "Enter last name: ";
 cin.get(last,6); cin.ignore(2000,'\n');             // it stops at first CR
 cout << first << " " << last << endl;
 strcpy(name,last);                                  // copy last[] into name[]
 strcat(name,", ");                                  // append a comma and a space
 strcat(name,first);                                 // append first[] to name[]
 cout << "Customer: " << name << endl;
 cout << first << " " << last << endl;               // "just in case"
 return 0;
}
```

```
Enter first name: John
Enter last name: Smith
John Smith
Customer: Smith, John
John n
```

图5-7 在串接中导致数组溢出，并无声地破坏了其他数据

在这里，数组first[]含有"John"，而数组last[]含有"Smith"；这些数组在输入时被保护而防止溢出。然后，在数组name[]中，作者将名字的各项串接起来并得到"Smith, John"；这个字符串包括0终止符在内共含有12个字符。由于数组name[]只有10个字符，因此后面两个字符"n\0"会被放在其他地方。在作者的计算机上，它们被放在了数组last[]中。结果，数组last[]含有这两个字符以及从"Smith"那里余下的一些字符，也就是变为"n\0ith\0"。于是程序的最后一条cout语句在打印名字时，能够正确地输出；但当它打印姓氏时，输出了"n"之后就停止了。

为了处理这个问题，C++库string.h提供了函数strncpy()和strncat()，它们类似于strcpy()和strcat()，但给出了第三个参数，该参数指定了将要复制的字符个数。

它们的使用如程序5-8所示。当复制了指定的字符个数后（或到达第二个参数的结尾时），函数strncat()就会终止复制并添加空终止符。这是安全的。函数strncpy()只有在第二个字符串的长度短于指定长度时，才会添加空终止符，当达到限制时，它会终止复制但不会添加岗哨字符。因此，strncpy()不会总是产生一个合式的字符串，这样的使用是不安全的。图5-8表明使用strcat()确实能防止目标数组不会溢出，这时目标数组name[]含有被截取的数据（“Smith, Joh”而不是“Smith, John”），但这个数据是合式的。

程序5-8 在串接中防止数组溢出的一个例子

```
#include <iostream>
#include <cstring>                                    // notice a new header file
using namespace std;
int main()
{
 char first[6], last[6], name[10];
 cout << "Enter first name: ";
 cin.get(first,6); cin.ignore(200,'\n');              // it has to remove CR
 cout << "Enter last name: ";
 cin.get(last,6); cin.ignore(200,'\n');               // it stops at first CR
 cout << first << " " << last << endl;
// strncpy(name,last,4);                              // no null if length>=count
 strcpy(name,last);
 cout << "After copy: " << name << endl;
 strcat(name,", ");
 strncat(name,first,3);
 cout << "Customer: " << name << endl;
 cout << first << " " << last << endl;                // "just in case"
 return 0;
}
```

```
Enter first name: John
Enter last name: Smith
John Smith
After copy: Smith
Customer: Smith, Joh
John
```

图5-8　截取数据以防止内存破坏

问题是我们没有正确地计算出截取的字符数，数组name[]有10个字符，而字符串"Smith, Joh"包括0终止符在内共含有11个字符。终止符去了哪里呢？在作者的计算机上，它使数组last[]的第一个字符变为终止符，原有的内容被破坏为“\0mith”，而不是"Smith"。当最后那个cout打印数组last[]时，发现终止符是第一个字符，于是没有打印任何内容。

提示　对于所有在字符串上进行的操作，应保证较长的输入数据不会溢出字符数组，并不会破坏与其不相关的程序数据。这些错误难以发现，因为被破坏的是不相关的数据

C++程序员有着非常大的影响力，他们编写的程序会影响很多人。他们所用的语言是强大而优美的。但对于没有经验的新手来说，该语言是危险的，应该保证语言被正确地使用。

5.1.9　二维字符数组

一个字符数组可以存放多个单词。对于很多进行文本处理的应用问题，要将文本拆分为一系列单词并组织成单词数组来处理。这时可以把数组看成是一个char类型的二维字符数组。例如，一个表示一周中每天的数组。我们要对数组day[]检查其输入数据中是否含有："Sunday"或"Monday"或"Tuesday"等等。虽然这些单词长度不等，我们还是把这些数据

表示为由7行组成（每天一行）的二维数组。这些行必须等长。最大长度的单词是有9个字符的单词"Wednesday"，加上另一个存放终止符的位置，这个数组必须有10列。

```
char days[7][10], day[10];
```

如果我们想检查数组day[]里是否存放着一周中的某天，就把这个数组与数组days[][]的每一行进行比较。可以逐个地比较第i行（i从0变到6）的每一个元素。如果数组days[]的字符与数组days[][]第i行的字符一样，我们就在下标i处停止循环。由于已在数组中找到了输入数据，因此，在外层循环必须了解内层循环的情况，于是通常会用一个控制标志（例如，变量found）来控制循环。在内层循环开始之前，我们把found设置为1（真）；如果内层循环发现了不同的字符，就把found设置为0（假），从而向外层循环表明该单词还没有找到。如果数组day[]中的所有字符和数组days[][]的第i行完全一样，就永远不会执行found=0;这一赋值语句，标志的值仍然是1，于是由break语句终止外层循环。

```
for (i = 0; i < NUM; i++)
{ found = 1; j = 0;
  do {
    if (day[j] != days[i][j])          // word is not found
     { found = 0; break; }             // stop, do it for next i
       j++;
      } while (day[j] != '\0');
  if (found == 1) break; }             // break outer loop
```

一些C++程序员将把后4行编写得更加简洁：

```
do
{ if (day[j]!=days[i][j++])            // compare and increment
   { found = 0; break; }               // stop, do it for next i
    } while (day[j]!='\0');            // no need for separate j++
   if (found) break; }                 // any nonzero value is true
```

使用条件语句的简化形式是安全且合适的。然而，把比较运算和加1运算结合在一起是不顾后果的。在这里j被用在两个子表达式中，正如第3章中所述，我们不知道它们的求值顺序。在作者的计算机上，这段代码的执行不正确。在有些计算机上，它可能会正确执行。要注意的是：一个谨慎的C++程序员一定会意识到这个问题，并避免用这种编码风格。

通常，我们通过使用库函数strcmp()来简化处理。程序5-9给出的程序提示用户输入一周中的某天，然后在二维字符数组中进行搜索，显示查找到的某天。注意，在搜索循环终止之后，程序必须确定循环为什么会终止，是因为该单词被找到了，还是因为查找完数组的所有单词之后都没有找到与之匹配的单词。为此，有几种做法。在此所用的是检查索引i的值，如果它等于数组中元素的个数，就意味着搜索不成功。如果它小于元素个数，循环就是由break语句终止的。图5-9给出了该运行的结果。这里没有测试所有可能的合法输入数值，这是不谨慎的：因为在数组初始化时，"Friday"被拼写错了。

程序5-9 对二维字符数组进行搜索

```
#include <iostream>
#include <cstring>
using namespace std;
#define NUM 7                              // do we expect the week length to change?

int main(void)
```

```
{
 int i; char day[10];
 char days[NUM][10] = { "Sunday","Monday","Tuesday",
      "Wednesday","Thursday","friday","Saturday" };
 do {                                                  // do until the user enters "end"
  cout << "Enter day of the week or 'end' to finish: ";
  cin.get(day,10); cin.ignore(2000,'\n');              // this is prudent
  cout << "Your input: " << day << endl;
  if (strcmp(day,"end")==0) break;
  for (i = 0; i < NUM; i++)
   if (strcmp(day,days[i])==0) break;                  // stop search if found
  if (i == NUM)                                        // check why we got here
  cout << "Input \"" << day << "\" is incorrect\n";
 else
  cout << day << " is day no. " << i+1 << endl;
 } while (1==1);                                       // go on forever
 cout << "Thank you for using this program" << endl;
 return 0;
}
```

```
Enter day of the week or 'end' to finish: Sunday
Your input: Sunday
Sunday is day no. 1
Enter day of the week or 'end' to finish: Thursday
Your input: Thursday
Thursday is day no. 5
Enter day of the week or 'end' to finish: saturday
Your input: saturday
Input "saturday" is incorrect
Enter day of the week or 'end' to finish: end
Your input: end
Thank you for using this program
```

图5-9　在测试中走捷径会引起麻烦

5.1.10　插入算法中的数组溢出

程序5-9中的算法只是对数组数据（一周中的各天）进行搜索，而数组的溢出问题并没有出现。该数组的大小等于数组中有效元素的个数。通常，数组在进一步处理之前必须先填入数据，这些数据占用数组的前面部分，而每一个新元素都添加在当前最后一个元素之后。一般情况下，可用数据的数组部分是一组连续的元素，而数组的后面部分是可用的位置，但它们没有存放有效的数据。

应该对存放着有效数据的连续部分进行进一步处理，而不是对该数组的每一个元素进行处理。因此经常需要记录数组中究竟有多少个有效的数据元素。对数组添加（或插入）元素的算法必须关心数组的溢出问题。

程序5-10给出了输入处理数据的算法。这是曾在第4章讨论过的那些例子的扩展。在那些例子中使用一个伪负数表示输入的结束。在这里，我们采用另一个做法：请用户输入“end”来终止输入。（通过按Enter键来指示输入结束并不是不好，但可能会在输入的过程中不小心按下该键）。为了能够把输入数据既当做文字也当做数字，程序把数据输入到数组buff[]里并检查当前的输入是否是“end”标记。如果不是该标记，程序就调用在头文件cstdlib

(或stdlib.h)中定义的库函数atof(),把字符串转化为浮点数。在这个函数名中,'a'代表ASCII、'to'代表"转换为"、尽管希望'f'代表float,但实际上它代表了double。另外,还有函数atoi()(ASCII转换为整数)和atol()(ASCII转换为long),但没有atod()函数。这些函数最多可以处理100个字符,所以一个有20个字符的缓冲是合适的。如果缓冲区含有非数值数据,atof()就会返回0并且程序会对用户发出警告。一个更好的策略是在输入数据中,检查除了数字之外是否还含有其他内容,但这就要用到函数strtod()和strtol()(串转换为double和long)并使用指针。

程序5-10 将处理的数据存入连续数组

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

int main ()
{
 const int NUM = 100;                              // the size can of course change
 double total, amount, data[NUM]; int count;
 char buff[20];
 total = 0.0; count = 0;                           // initialize current data
do {                                               // do until the user enters "end"
 cout << "Enter amount (or 'end' to finish): ";
 cin.get(buff,20); cin.ignore(2000,'\n');
// cout << "You entered '" << buff << "'" << endl;  // debugging
 if (strcmp(buff,"end")==0) break;
 amount = atof(buff);                              // convert to double up to 100 chars
// cout << "Amount: " << amount << endl;            // debugging aid
 if (amount <= 0)                                  // zero if non-numeric input
   cout << "This value is discarded as incorrect.\n"
        << "Please reenter it correctly.\n";
 else
  { total += amount;                               // process current data
    data[count] = amount;
    count++; }
 } while (1 == 1);
 cout << "\nTotal of " << count << " values is "
      << total << endl;
 if (count == 0) return 0;
 cout << "\nTran no. Amount\n\n";
 for (int i = 0; i < count; i++)
  { cout.width(4); cout << i+1;
    cout.width(11); cout << data[i] << endl; }
 return 0;
}
```

否则,程序累加total的值,将输入数值保存在数组中,并使下标值加1。在输入结束时,变量count将会含有数组中有效数值的个数。因此循环(打印该处理值)会直到下标到达count而不是等于数组元素个数NUM才停止。注意库函数width()的使用,它规定了在输出中为下一个数值所分配的最小宽度。其缺省宽度是0:该值只占有输出数据所需的宽度。如果输出中的字符个数小于所需的宽度,剩下的位置就用空格来填充(数字向右对齐,字符串向左对齐)。如果要显示的字符个数大于所需的宽度,就会忽略宽度规定,而按照所需的宽度输出。

请注意对以下语句的测试，这些语句显示输入数据并且将它们转换为数值形式。当程序不能正确地解释输入数据时，就会经常出错。此时，检验程序实际从输入中得到什么是一个好方法。该程序的输出如图5-10所示。

```
Enter amount (or 'end' to finish): 22
Enter amount (or 'end' to finish): 33
Enter amount (or 'end' to finish): 44
Enter amount (or 'end' to finish): 55
Enter amount (or 'end' to finish): end

Total of 4 values is 154

Tran no.  Amount

   1         22
   2         33
   3         44
   4         55
```

图5-10　使用连续数组来存放输入数据

程序5-10的程序有很好的运行结果，它使用连续的数组元素来存贮数据。第二个循环没有显示所有可用的数组元素，只显示了通过第一个循环存入数组的那些元素。然而，如果输入数值的个数超过了数组长度，该程序并没有防止数组溢出。

为了防止数组溢出和内存破坏，第一个循环应该测试下标count是否指向一个合法的数组元素。记住，第一个合法的下标值是NUM，即数组中元素的个数。因此，只要count小于NUM，数值就可以保存在数组中；否则，输入必须终止。

实际的程序中会含有较长的数组，并且为了测试是否能够防止溢出，必须提供几百个或几千个数值。这会浪费时间并容易出错。程序5-11给出了对程序5-10的修改，它能够防止溢出。为了避免使用大量数据，将数组的大小减小为3，这是一个好的测试技巧。

在前一节的例子中，字符数组没有被修改，其下标值用于说明循环为何终止。如果下标值小于元素个数，就意味着找到了所需项目；否则，表示已经测试了所有的数组元素而查找失败。在这个例子中，该方法并不是完全可靠的，因为当用户刚好输入NUM个值时count也达到了NUM。对于一个很大的数组来说，这是很罕见的。因此一些程序员并不介意在代码中使用这种方法，但这是不正确的。程序5-11测试用户是否输入了“end”，如果没有，循环就会由于数组溢出而终止。其执行结果如图5-11所示。

程序5-11　能够防止溢出地将数据输入数组

```
#include <iostream>
#include <cstring>
#include <cstdlib>                          // to support atof()
using namespace std;

int main ()
// { const int NUM = 100;                   // the size can of course change
{
  const int NUM = 3;                        // for debugging only
```

```
  const char LAST[] = "end";                    // literal for termination
  double amount, total, data[NUM];
  char buff[20]; int count;
  total = 0.0; count = 0;                       // initialize current data
do {                                            // do until the user enters "end"
  cout << "Enter amount (or '" << LAST << "' to finish): ";
  cin.get(buff,20); cin.ignore(2000,'\n');
  if (strcmp(buff,LAST)==0) break;              // end of input data
  amount = atof(buff);                          // convert to double up to 100 chars
  if (amount <= 0)                              // zero if non-numeric input
  cout << "This value is discarded as incorrect.\n"
       << "Please reenter it correctly.\n";
  else if (count < NUM)
  { total += amount;                            // process current data
    data[count] = amount;
    count++; }
  else
  { cout << "Out of memory: input is terminated\n";
     break; }
  } while (1 == 1);
  if (strcmp(buff,"end") != 0)
      cout << "The value " << amount << " is not saved\n";
  cout << "\nTotal of " << count << " values is "
       << total << endl;
  if (count == 0) return 0;
  cout << "\nTran no. Amount\n\n";
  for (int i = 0; i < count; i++)
   { cout.width(4); cout << i+1;
     cout.width(11); cout << data[i] << endl; }
  return 0;
}
```

```
Enter amount (or 'end' to finish): 22
Enter amount (or 'end' to finish): 33
Enter amount (or 'end' to finish): 44
Enter amount (or 'end' to finish): 55
Out of memory: intput is terminated
The value 55 is not saved

Total of 3 values is 99

Tran no.  Amount

   1         22
   2         33
   3         44
```

图5-11 使用一个较短的数组来测试能否防止溢出

这段代码也给出了与数组LAST[]一起使用的关键字const。这使得维护人员很容易将终止符改变为"finish"或一个空字符串或其他字符，而不必在源代码中跟踪"end"的所有出现。对于符号常量NUM也有一样的道理。改变一个常量的值比进行全局修改要好得多。如果要把数组大小从100改为300，就可以使用全局的replace命令，这是一个节约开销的做法。

请注意，如果不把头文件cstdlib（或stdlib.h）包含进来，编译程序将认为atof()的调用是语法错误，并告诉我们它不知道atof()函数是什么。实际上编译程序知道库函

数是什么和它们在哪里，如果我们不记得调用atof()需要什么头文件时怎么办呢？去问编译程序吧。在UNIX下，我们可以输入命令man atof。在Windows下，我们可以在源代码中选择atof并点击F1以寻求帮助。帮助页面会立即显示出来，它会告诉我们所有与atof()有关的东西，包括其头文件的名字。

5.1.11 数组类型的定义

在本章前面的所有例子中，我们所使用的数组是数组变量而不是数组类型。如果我们需要另一个与给定数组结构相同的数组（也就是有相同的类型和元素个数），就必须重新定义另一个数组。

```
double data[NUM];
double tax[NUM];
```

如果这些定义出现在程序的不同地方，维护人员（或设计人员）就需要花费额外的精力，才能明白原来这两个对象有着相同的结构。

一个更好的表达方法是定义一个类型，比如定义一个有NUM个double类型元素的数组SalesData，于是可以使用这个类型名去定义属于该类型的任意数组的变量。

```
SalesData data;
SalesData tax;
```

这种方式与使用固有的标量类型去定义属于该类型变量的方式是完全一样的。C++提供了关键字typedef来实现这种处理。通常，typedef是一种基于编译程序已知的类型名来定义新的名字(包括类型名)的机制，其一般的形式是将已知的类型名与等价于该类型的新名字写在一起：

```
typedef known_type new_type_name;
```

定义了这条语句之后（以分号结束），程序就可以把know_type与new_type_name看做同义词。

下面这段处理库存信息的代码是使用typedef的一个简单例子。

```
int idx, quant, const MAX=30, qty[MAX];
for (idx = 0; idx < MAX; idx++)
  { cin >> quant;
    qty[idx] = quant; }
```

在这里，变量idx、quant、MAX以及数组qty[]的元素都是整数，然而它们是不同性质的整数：一个是数组下标，而其他描述库存项目的数量。“qty[idx]=quant;”这样的语句是有意义的；而语句“idx=quant;”就没有什么意义。就C++的规则而言，两个语句都是合法的。为了强调它们的性质不同，还可以使用的方法就是引入两个新的类型名。

```
typedef int Index;                          // one kind of integers
Index idx;
const Index MAX=30;
typedef int Stock;
Stock quant, qty[MAX];                      // another kind of integers
for (idx = 0; idx < MAX; idx++)             // comparison between the same type
  { cin >> quant;
    qty[idx] = quant; }                     // assignment between the same type
```

这里，idx和MAX都属于同一种类型Index，因此对它们进行大小比较是合法的。变量quant和qty[idx]属于同一种类型Stock，因此赋值运算是合法的。但如果程序员写出

"idx=quant;"，那么这个赋值运算是不可靠的，因为这两个变量属于不同的类型。

在这个例子中，使用typedef为已存在的类型int定义了一个新的名字，而不是引入一个新的类型。只有对程序员来说两个变量才是不同的；对于编译程序来说，Index和Stock是同一个类型名的别名。

在定义一个数组类型时，我们可以使用typedef的另外一种格式来定义新类型：

```
int const MAX = 30;
typedef double SalesData[MAX];
```

在这个定义中，关键字typedef位于一个语法完整的数组定义之前。如果没有typedef，这个定义将引入一个新的名字SalesData，通过类型double和常量MAX把它定义为一个数组变量的名字。由于给出了typedef，这个定义就使新名字ScalesData成为了一个新类型的名字。根据typedef后面的定义，这个类型是一个有MAX个double类型元素的数组。（当然，MAX可以是任何一个编译常量，包括整型常量。）

现在我们可以使用这个类型名去定义属于该类型的变量。尽管变量的定义只需给出类型名和变量名，每一个变量都是一个有着MAX个double类型元素的数组。以下两个数组定义都为属于double类型的MAX个元素分配了内存空间。

```
SalesData data;
SalesData tax;
```

通过使用与一般数组变量同样的形式，变量data和tax可以与任意数组元素一样使用：

```
for (int idx = 0; idx < MAX; idx++)
  { tax[idx] = data[idx] * 0.05; }
```

尽管我们用来定义类型Index和Stock所用的typedef的形式，与用来定义类型ScalesData的形式不同，但它们的工作方式是一样的。它们把未经typedef语句定义的名字定义为一个新的类型名（比如第一个例子中的Index和Stock，以及第二个例子中的ScalesData）。

下一节，我们将讨论更多有关typedef的使用问题。

5.2 不同种类聚集的结构

在我们的讨论中，第二种程序员定义的数据类型是结构。C++结构是一个将相关的元素结合起来的强大的聚集工具。在C++中定义结构的方法不止一种，而我们将讨论那些最流行的方法。所有的方法都允许程序员定义结构的元素（域或数据成员），也就是指出元素的类型和名字。

5.2.1 程序员定义类型的结构定义

结构的定义以关键字struct开头，其后面跟着程序员定义的名字，该名字将在程序中作为一个类型名字来定义变量。结构的域声明位于花括号之内，且在花括号后面跟着一个分号。每一个域的声明都类似于一个变量的声明：它给出了类型名以及由程序员定义的域名，但不能进行初始化操作。

```
struct Account {                    // 'Account' is a type name now
  long number;                      // 'number' is a field name
```

```
  double balance;
  double overdue; } ;              // semicolon follows the brace
```

每一个域声明都以分号结束。如果在结构的定义中相邻的域都属于同一种类型，那么可以只用一个类型名并以逗号作为分隔符来进行这些域的声明。以下这个结构定义的结果与前一个是完全一样的：

```
struct Account {                        // 'Account' is a type name now
  long number;                          // 'number' is a field name
  double balance, overdue; } ;
```

C++还支持另一种从C继承下来的定义结构的语法，就是使用typedef定义一个新类型。

```
typedef struct tagAccount {
  long number;
  double balance, overdue; } Account;
```

这个形式与我们在前一节里用typedef定义整数的形式是一样的。

typedef 已知类型 新类型名;

这里，已知类型表示为struct tagAccount的定义，Account是新类型名（类似于前面使用typedef的所有例子，名字Account是未经定义的惟一的名字）。实际上，struct tagAccount也是一个类型名，它也可以用在Account所使用的任何地方；但使用这个名字会不大方便，因为使用了关键字struct，使得这个类型名在形式上不同于其他基本类型的名字。这种定义结构类型的形式在C中非常流行，但在C++中并不是必需的。

5.2.2 创建和初始化结构变量

结构的定义并没有分配任何内存。它为将来的内存分配定义了一个模板：要分配多少内存，怎样去解释该内存以及使用什么名字来存取内存中的变量。使用结构名进行变量定义的方式与使用其他基本类型，如int、double的方式是一样的。

```
Account a1, a2;          // memory for two Account variables
```

这里创建了两个Account类型的变量，每一个都有域名分别为number、balance和overdue的三个域。每一个Account类型变量的大小等于一个long类型和两个double类型值之和再加上校正空间（如果一个值不能从任意地址开始，就必须和能整除4或8的地址对齐）。

应该注意前一节中定义结构类型时所使用的花括号，它表示了一个块，像其他域一样具有单独的作用域（第6章会对这个内容进行更多的介绍）。在该作用域内定义的名字在作用域之外是不可知的。由于number是一个long类型的数据成员而不是一个long变量，我们不能没有任何限制地使用它的名字。

```
number = 800123456L;           // there is no such thing as number
```

这是没有意义的，因为不能确定这个number可以参加什么运算。C++圆点运算符或称选择运算符（属于高优先级）选择结构变量的域。类似于利用下标来访问数组元素，使用圆点运算符可以将结构域作为左值或者右值来访问。

```
a1.number = 800123456L;              // field is used as lvalue
if (a1.number == 800123456L)         // field is used as rvalue
  a2.number = a1.number;             // both lvalue and rvalue
```

类似于数组，一个结构中域的个数必须在编译时确定。数组元素属于同种类型，它们没有单独的名字，但它们是有顺序的。使用数组变量的名字和元素的下标可以引用数组元素，其结果是数组定义中所指定的类型的某个值。结构的元素是无顺序的，它们有着自己单独的名字并且可以属于不同的类型。使用结构变量的名字以及域的名字可以引用结构的元素，其结果是在结构定义中所指定的类型的值。

由于结构的域是无顺序的，因此Account的域可以按任意顺序进行定义，这不会改变使用该结构的程序的含义。

当创建结构变量时，它们的域并不含有任何有用的值。C++提供了类似于数组初始化的初始化语法，可以为每一个结构元素指定适当类型的值。

```
Account a1 = { 800123456L, 532.84, 0 } ;
```

这个语法只对于具有公共域的结构有效，客户代码可以存取这些结构的域（我们在本章讨论的结构都具有公共域）。不久，我们将学习怎样使用构造函数去初始化具有非公共域的结构和类变量。

类似于非聚集变量，利用以前已定义的另一个结构变量去初始化一个结构变量是允许的。

```
Account a3=a1;              // it has 800123456, 532.84, 0 in fields
```

5.2.3 层次结构及其分量

使用C++结构的目的是支持数据抽象和封装。结构的域代表了与应用相关的对象的属性：例如个人数据、医疗记录、库存数据和顾客数据。它们也代表了一些经常一起使用的相关信息：任务控制块、分析程序符号表、字体阵列结构、通信包等等。结构在系统程序设计和应用程序设计中使用得非常普遍。

每一个结构变量表示一个复合对象，其分量可以单独使用也可以作为整体来使用。结构变量可以作为一个单元进行处理，并被传递给函数。在函数中，这些分量可以单独处理，可以更进一步用结构变量组成数组、链表、队列等等。

```
Account cards[500];                 // array of 500 structures
```

当我们访问属于这个数组元素的域时，必须使用层次符号。下标运算符和圆点选择运算符属于同一个优先级，并且从左向右结合。以下就是访问数组cards中下标为75的元素的number域的方式：

```
cards[75].number = 800123456L;
```

当然，我们也可以把结构作为其他结构的分量。例如，我们可以把顾客名字、地址和账目数据组合为一个新的类型。

```
struct Customer
  {
    char name[30]
    char address[70];
    Account acct;
  } ;
```

在这个格式中，每个括号和每个域都独占一行，以便显示该结构的形式。其他程序员可以选择更简洁的格式：

```
struct Customer
  { char name[30], address[70];
    Account acct; } ;
```

当创建一个Customer变量时，其内存中包括了两个字符数组和一个含有一个long域和两个double域的结构变量Account。同样，访问数组的元素和Account的分量也要使用层次符号。

```
Customer c;
strcpy(c.name,"Doe, John");                          // c.name is of type char[]
strcpy(c.address,"72 Main, Anytown, MA");
c.acct.number = 800123456L;                          // type long int
c.acct.balance = 532.84; c.acct.overdue = 0;         // type double
```

同样，圆点选择运算符从左向右结合，并且应该从右向左阅读。例如c.acct.balance是acct变量（类型是Account）的balance域（类型是double），而acct又是结构变量c（类型是Customer）的域。

大家可以看到这很容易变得相当冗长和笨重。处理这个问题的一个方法是，编写那些简化处理聚集变量代码的访问函数。在后面几章中，我们将会看到大量访问函数的例子。

5.2.4 结构变量上的操作

如果结构变量属于同种类型，那么它们可以互相赋值。源变量的域值将被逐个拷贝到目标变量的域中。

```
a2 = a1; c.acct = a1;            // same type (Account)
```

这等价于下面的一组赋值操作：

```
a2.number = a1.number;
a2.balance = a1.balance; a2.overdue = a1.overdue;
c.acct.number = a1.number; c.acct.balance = a1.balance;
c.acct.overdue = a1.overdue;
```

这里，C++的强类型性质非常明显。不同结构类型之间不允许做任何转换。

```
c = a1; a1 = c;              // no, they are of different types
a1 = 800123456L;             // do not even think about it!
```

这里的问题与类型的名字有关而与结构的构成无关。假设有一个与Account的构成相同的结构：

```
struct FrozenAcct
{ long number;                       // same structure as Account
  double balance, overdue; } ;
```

一个FrozenAcct结构变量仍然不能被赋值给一个Account结构变量，反之亦然。

```
FrozenAcct fa; fa = a1;          // no, type names are different
```

在C++中没有结构的比较运算，因为C++不知道应该使用什么域来比较以及怎样进行比较。如果需要比较结构变量，就必须编写代码去满足这一应用要求。

```
if (a1.number > a2.number)           // swap accounts to order numbers
 { a3 = a1; a1 = a2; a2 = a3; }      // a3 holds data temporarily
```

下一个例子给出了一些图形计算。由于画图函数是不可移植的，因此例子中没有作图，

而是要求用户输入两条线段端点（AB和CD）的坐标，然后计算每条线段的长度以及每条线段与x轴之间的夹角。

程序5-12给出了实现这个处理的源代码，它使用了两个数据类型Point和Line。该程序要求输入以像素为单位的数据（整数），使用这些数据去初始化端点，使用这些端点去初始化两条线段，然后计算线段的长度和夹角。函数sqrt()和atan2()来自math.h库头文件，它们计算其double类型参数的平方根和反正切值。因为它们的实际参数被定义为int，于是它们被隐式地转换为double类型。由于线段的长度以像素为单位，也是整数，计算平方根的结果被隐式地转换。为了避免截取的丢失，对该长度加上0.5以获得适当的舍入。变量coeff把角度从弧度转换为度数。该程序的执行结果如图5-12所示。

程序5-12 使用#include指令的程序员定义类型

```
#include <iostream>
#include <cmath>                          // to support sqrt() and atan2()
#include "point.h"                        // to make type Point known to compiler
#include "line.h"                         // to make type Line known to compiler
using namespace std;

int main ()
{
 const double coeff = 180/3.1415926536;
 Point p1, p2; Line line1, line2;         // programmer-defned types
 int diffX, diffY, length1, length2;
 double angle1, angle2;
 cout << "Enter x and y coordinates of point A: ";
 cin >> p1.x >> p1.y;
 cout << "Enter x and y coordinates of point B: ";
 cin >> p2.x >> p2.y;
 line1.start = p1; line1.end = p2;
 cout << "Enter x and y coordinates of point C: ";
 cin >> p1.x >> p1.y;
 cout << "Enter x and y coordinates of point D: ";
 cin >> p2.x >> p2.y;
 line2.start = p1; line2.end = p2;
 diffX = line1.end.x - line1.start.x;  // ugly notation
 diffY = line1.end.y - line1.start.y;
 length1 = sqrt(diffX*diffX + diffY*diffY) + 0.5;
 angle1 = atan2(diffY,diffX) * coeff;
 cout << "Length AB is " << length1 << " at angle "
      << angle1 << " degrees\n";
 diffX = line2.end.x - line2.start.x;
 diffY = line2.end.y - line2.start.y;
 length2 = sqrt(diffX*diffX + diffY*diffY) + 0.5;
 angle2 = atan2(diffY,diffX) * coeff;
 cout << "Length CD is " << length2 << " at angle "
      << angle2 << " degrees\n";
 return 0;
}
```

5.2.5 在多文件程序中定义的结构

对于像程序5-12那样的小例子来说，把结构的定义写在源代码中是完全合理的。由于在

程序的名字空间中程序员定义的类型必须是惟一的，因此，如果一个程序员定义的类型要用在若干个文件中，那么一个多文件程序就可能产生问题。实际上，程序员不会喜欢在几个文件中重复该类型的定义。如果这些定义在维护过程中经常更新，它们就很容易变得不一致。

```
Enter x and y coordinates of point A: 20 20
Enter x and y coordinates of point B: 80 80
Enter x and y coordinates of point C: 20 20
Enter x and y coordinates of point D: 160 80
Length AB is 84 at angle 45 degrees
Length CD is 152 at angle 23.1986 degrees
```

图5-12　程序5-12中的程序的执行结果

解决的方法是把每个类型的定义放在一个单独的头文件里，并把这些文件包含到使用该类型的所有源文件中，就像程序5-12中的Point和Line类型那样。注意对文件名使用的是双引号而不是尖角括号。与在文件中定义的类型一样，头文件通常会使用同样的名字命名。由于C++程序中所有的类型名必须惟一，因此，当把头文件置于相同的目录下时不会引起名字冲突。通常，头文件存放在一个单独的目录中，它不同于可执行程序文件的目录。在这种情况下，#include指令应该指定完整的路径名。

当同一个程序员定义的数据类型用在一个多源文件程序中时，为了避免该数据类型的重复编译，一个常见的做法是使用条件编译。程序5-13和程序5-14给出了该头文件的内容。

程序5-13　头文件 "point.h"

```
#ifndef _POINT
#define _POINT
struct Point
{ int x, y; } ;
#endif
```

程序5-14　头文件 "line.h"

```
#ifndef _LINE
#define _LINE
#include "point.h"
struct Line
{ Point start, end; } ;
#endif
```

当我们把这些文件包含到几个源文件中时，编译程序处理的第一个定义就是定义名字_POINT和_LINE。当编译其他文件时，这些文件中的条件编译指令会使这两个类型定义不被编译。对于作为类型名的符号常量，人们经常会使用相同的名字，因此要使用大写并在其前面加上下划线来避免可能发生的命名冲突。

注意文件"line.h"必须包含文件"point.h"。否则，编译程序可能会不知道在文件"line.h"中的类型名Point的含义是什么。

以上就是灵活使用结构时程序员所需要知道的全部内容。使用结构时最主要的一点是只能把相关的信息放到结构中，要避免包含与其他域不相关的信息。另外要注意的是，通过使用圆点运算符以及在结构定义中声明的名字，可以访问每个结构变量的域，每个域（用圆点

运算符表示）都是在结构定义中属于指定类型中的一个变量。

5.3 联合、枚举和位域

这一节的内容相对简短。它将讨论为了方便程序员而对程序实体命名的三种思想。第一种思想是定义一个可以用来存储多于一种类型信息的变量，例如存储一个整数和一个浮点数，这就是联合（union）。第二种思想是为相关的常量定义符号名而不用关心将数字值赋值给这些符号常量的细节，这就是枚举（enumeration）。第三种思想是对名字的截取，以便可以对名字抽取其部分的内容进行操作，这就是位域（bit field）。

C++以一种类似于定义结构的方式来实现这些思想。程序员在类型定义的开头使用一个关键字（union、enum或struct）。然后，程序员为这个新类型引入新的名字并描述该类型的构成（位于花括号内，以分号结束）。定义了新类型以后，这一程序员定义的类型名就可以作为一个类型名在程序中使用。

5.3.1 联合

让我们考虑一个Number类型的结构数组，它含有某些任意数字信息。

```
Number num[6];
```

由于对该数组来说，任何数字值都是有效值，因此可以使用一个非数字值作为标记，例如一个字符串"end"或其他东西，但C++的强类型不允许在一个数字域里存储文本信息。对这个问题的一个解决方法就是定义一个具有两个域的结构，一个域存放数字值，另一个域存放文本，形如以下的定义：

```
struct Number
{ double value;
  char text[4]; } ;
```

现在我们可以使用属于这个类型的数组元素，把数字值信息放在value域中，在text域中存放标记，比如用"end"表示数组中有效数据的结束。然而，每个数组元素只使用了一个域；它是一个有效的数字或者是文本标记。通过定义一个联合，C++使我们可以避免空间的浪费。在C++中，用关键字union为内存的相同区域提供两个或多个不同的解释。关键字union使用与结构定义相同的语法来定义新的类型。如果有几个域，它们就表示了对内存的相同区域的多种解释。在这一例子中，可以用union定义如下：

```
union Number                    // yet another C++ keyword
{ double value;                 // any number of fields can be defined
  char text[4]; } ;             // do not forget the semicolon!
```

这个定义引进了类型Number。我们可以定义属于这个类型的变量，并且每个变量将有两个域。与结构不同的是，这些域不会同时存在，它们代表着对内存的相同区域的不同解释。当定义一个联合变量时，会分配足够的空间去容纳最长的解释。程序员可以选择使用哪种解释，是解释为浮点数还是字符数组。如果不小心将数据作为某一种类型的数据存放，然后又当做另一种类型的数据读取，那么其结果将是垃圾。与其他情况一样，C++编译程序不会帮程序员检查对内存的使用是否一致。

下面这段样本代码是将一个数字值存放在联合变量n1中，并将一个字符串存放在联合变

量n2中。但随后却把n1的内容显示为文本而把n2的内容显示为数字值，这就得不到有用的结果，但编译程序并不会阻止我们这样做。这个例子也说明了可以合法地在存储数字值的地方存储文本，反之亦然。

```
Number n1, n2;
n1.value = 5.0; strcpy(n2.text,"no");          // making a commitment
cout << n1.value << " " << n2.text << endl;    // this is OK
cout << n1.text << " " << n2.value << endl;    // this is a disaster
strcpy(n1.text,"yes"); n2.value = 25.0;        // old contents disappears
cout << n1.text << " " << n2.value << endl;    // now this is OK
```

这个例子说明当使用联合变量时，其形式与结构类似。程序5-15给出了说明这一点的一个小程序。看起来好像数组num[]的前三个元素的text域没有被初始化，并且最后一个元素的value域没有被初始化。实际上，这里同一个区域被用作两种解释。（编译程序为每一个元素分配相同数量的空间，它足以容纳最长的解释。）该程序的输出结果如图5-13所示。

程序5-15　在一个变量中使用union来存储几个不同类型的值

```
#include <iostream>
#include "number.h"                            // to make type Number known to compiler
#include <cstring>
using namespace std;

int main ()
{
 Number num[6]; int i = 0;                     // array of union variables
 num[0].value = 11.0; num[1].value = 21.0;     // initialization
 num[2].value = 31.0; strcpy(num[3].text,"end");
 while(strcmp(num[i].text, "end") != 0)        // iteration
   cout << num[i++].value << endl;
 cout << num[i].text << endl;                  // for illustration purposes
 cout << "Text as double: " << num[i].value << endl;

 return 0;
}
```

```
11
21
31
end
Text as double: 3.57452e-031
```

图5-13　程序5-15中程序的输出结果

对类似于num[0].value这样的符号，我们不要觉得难以理解。数组num[]含有属于Number类型的元素，因此num[0]属于Number类型并具有value和text两个域。当使用text域时，它是一个字符数组，因此可以把num[i].text作为一个参数传递给函数strcmp()。在输出域值时（如在num[i++].value中）使下标值加1也是合法的。由于在这里只使用了下标i一次，因此没有与求值顺序有关的危险。

程序的最后两行给出了对相同的值做不同解释时的情形。错误地使用联合很容易产生垃圾。注意，数值3.57452e-031有时在应用中可能是一个合法的浮点数值；它会被解释为

"end"并终止循环。

为了避免解释错误，有些程序员使用了所谓的标志域，利用其域值说明怎样去使用该联合类型。这个标志域不能是联合的成员，因此该联合和该标志域必然是一个更大结构的一部分。例如，地址可能含有3行，且第二行可能是街道地址或者是邮政编码。程序5-16给出了一个示例程序，它用一个联合域second和一个标志域kind来定义地址结构。当标志域是0时，第二行就被解释为街道地址；当标志域是1时，第二行就被解释为邮政编码。在设置数据（通过将kind的值设置为0或1）和使用数据（通过测试kind域的值）时，程序坚持这个约定。该程序的输出结果如图5-14所示。

程序5-16 使用带有标志域的union来加强存取的完整性

```
#include <iostream>
#include <cstring>
using namespace std;

union StreetOrPOB
  { char street[30];          // alternative interpretations
    long int POB; } ;

struct Address
{ char first[30];
  int kind;                   // 0: street address; 1: P.O.B.
  StreetOrPOB second;                          // either one or another meaning
  char third[30]; } ;

int main ()
{
  Address a1, a2;
  strcpy(a1.first,"Doe, John");                // address with street
  strcpy(a1.second.street,"15 Oak Street"); a1.kind = 0;
  strcpy(a1.third,"Anytown, MA 02445");
  strcpy(a2.first,"King, Amy");
  a2.second.POB = 761; a2.kind = 1;            // address with POB
strcpy(a2.third,"Anytown, MA 02445");
  cout << a1.first << endl;
  if (a1.kind == 0)                            // check data interpretation
    cout << a1.second.street << endl;
  else
    cout << "P.O.B. " << a1.second.POB << endl;
  cout << a1.third << endl;
  cout << endl;
  cout << a2.first << endl;
  if (a2.kind == 0)                            // check data interpretation
    cout << a2.second.street << endl;
  else
    cout << "P.O.B. " << a2.second.POB << endl;
  cout << a2.third << endl;
  return 0;
}
```

这样做的输出结果很好，但又给类型的层次结构引入了另一个层次。因此，程序员必须使用像a1.second.street这样的名字，这是不方便的。同时，在程序中，类型StreetOrPOB只是和类型Address在一起时使用了一次。为了克服这一缺点，C++支持匿名

联合。它们没有名字，并且不能定义这个类型的变量；然而，它们的域可以无任何限制地使用。例如，我们可以不使用类型StreetOrPOB而是使用一个匿名联合来定义Address类型。

```
Doe, John
15 Oak Street
Anytown, MA 02445

King, Amy
P.O.B. 761
Anytown, MA 02445
```

图5-14　程序5-16中程序的输出结果

```
struct Address
{ char first[30];
  int kind;                    // 0: street address; 1: P.O.B.
  union
  { char street[30];
    long int POB; } ;          // no 'second' field of type StreetOrPOB
  char third[30]; } ;
```

联合类型消失了，但类型Address现在有两个可选域street[]和POB，并且可以像其他域一样通过名字来引用它们。当然，选择使用哪一个是程序员的责任。数据的存取方式必须与它们的设置方式相一致，但这样做就不再需要另一个层次结构了。

```
if (a1.kind == 0)                    // check data interpretation
  cout << a1.street << endl;         // use one interpretation
else
  cout << "P.O.B. "<< a1.POB << endl;    // or use another one
```

这是一种很有用的程序设计技术。然而，维护人员必须花费额外的时间和精力去理解该代码。因为额外的条件语句增大了代码的复杂性。很可能，与虚函数一起使用的继承是这种程序设计技术的良好竞争者，不久我们将讨论到它。

5.3.2 枚举

枚举类型允许程序员从一组已定义的标识符中定义接受值的变量。通常，我们引入整型符号常量（使用#define或const定义）并建立使用它们的规则。例如，为了模仿交通灯的行为，我们需要表示灯的红、绿和黄等颜色值。类似于表示一周中各天的例子，我们可以引入字符数组"red"、"green"和"yellow"，并通过使用串操作库函数去进行赋值和比较操作。

```
char light[7] = { "green" };              // it is green initially
if (strcmp(light, "green") == 0)          // next it is yellow
  strcpy(light, "yellow");                // and so on
```

这就利于维护人员理解代码设计人员的意图，但这些字符串操作较慢。我们不会仅仅为了跟踪交通灯的状态，而去进行大量的字符移动（利用库函数搜索终止符）。这个方法的另一个缺点是缺少保护，如果有人想使灯变为粉红色或紫红色，没有办法阻止他这样做。

另一个解决方法是使用整数编号表示颜色。可以把0指定为绿、1指定为红、2指定为黄。注意这里是怎样规定这些值的：0、1和2，而不是1、2和3。这是C++数组以及下标从0开始计数的处理方式。

使用这个方法，我们就可以避免使用字符串处理函数。

```
int light = 0;              // it is green initially
if (light == 0)             // next it is yellow
    light = 2;              // and so on
```

这个方法的优点是速度快但仅限于此，因为这种编程形式总要求注解，特别是对于有更复杂的系统状态以及状态之间转换的复杂算法。如果其注解太隐晦或有点陈旧，要向维护人员转达设计人员的意图就不容易。

在保持速度的同时增强程序可读性的另一种方法是使用符号常量。我们可以定义那些其名字适合于应用的符号常量，例如RED、GREEN和YELLOW，并且可以用一个特殊的整数值赋给每个常量。

```
const int RED=0, GREEN=1, YELLOW=2;        // color constants
```

现在我们可以使用这些常量来重写上面的例子。该代码和前面的例子速度一样快，并且它和使用字符串的最初版本一样清晰。

```
int light = GREEN;                  // it is green initially
if (light == GREEN)                 // next it is yellow
  light = YELLOW;                   // and so on
```

这个解决方法保证了执行的速度，但它不保证在维护过程中代码不发生错误。如果维护人员（或原设计人员）使用号码代替符号常量，它不是语法错误。如果他们把颜色的允许取值范围之外的值（比如，light=42）赋给变量light，这也不是语法错误。我们还可以把这些值相加（比如，RED+GREEN），并且可以做一些实际上不会对颜色值进行的操作。

把枚举类型引入到语言中就是为了解决这种问题。程序员可以定义一种程序员定义类型并把该类型变量允许获得的所有合法值都显式地枚举出来。与使用关键字struct（或union）来引入程序员定义类型的方法类似，关键字enum用来引入程序员定义类型的名字（如Color），类型名后面跟着花括号（后跟着分号）。在花括号里，设计人员列出被定义的类型所允许的所有取值。通常，程序员使用大写（类似于#define和const的常量）对类型命名，但这不是必需的。对于这个例子，我们可以将类型Color定义为enum类型。

```
enum Color { RED, GREEN, YELLOW } ;        // Color is a type
```

现在，我们可以使用类型Color去定义那些只接受RED、GREEN和YELLOW值的变量。这些值是枚举常量：它们只能作为表达式的右值且不能被改变。

```
Color light = GREEN;                // it is green initially
if (light == GREEN)                 // next it is yellow
  light = YELLOW;                   // and so on
```

这个解决方法更完善了，在枚举类型上只定义了赋值操作和关系操作。我们不能将它们进行相加或进行输入输出，但可以比较它们是否相等，还可以检查一个值是否大于（或小于）另一个值。

```
if (light > RED) cout << "True\n";        // this prints 'True'
```

可以进行比较运算的原因是，在机器中枚举值是用整数实现的。在枚举清单中的第一个值是0（与C++计数的方式一样），下一个是1，等等。通过将枚举值转换为整数，程序可以存取这些值。

```
cout << (int) light << endl;          // this prints 0, 1, or 2
```

如果程序员想把这个值改为另一个值，可以在枚举程序中显式地实现：

```
enum Color { RED, GREEN=8, YELLOW } ;    // YELLOW is 9 now
```

在那之后，赋值运算的值就被重新设定（YELLOW是9，等等）。如果由于某种原因，我们想将GREEN设置为0，对编译程序而言这是允许的；但程序将不能够区分RED和GREEN的不同。

当枚举值用作按位操作的掩码以便代表2的幂时，这个技术就很有用：

```
enum Status { CLEAR = 2, FULL = 8, EMPTY = 64 } ;
```

很多程序员非常喜欢这个技术，使用它来定义编译时的整数常量。

```
enum { SIZE = 80 } ;                  // use it to define arrays etc.
```

注意，这个枚举是匿名的（类似于匿名的联合类型）。它没有名字，因此我们不能定义属于这个类型的变量；但这并不重要，因为我们所需要的是符号常量SIZE。其结果与显式定义常量一样。

```
const int SIZE = 80;                  // same thing
```

使用什么方法来定义常量没有统一的规定，每个人都可以选择自己喜欢的方法。

5.3.3 位域

类似于对联合和枚举的讨论，我们从一个使用C++的用户自定义的类型来解决实际问题的例子开始。

在C++程序中可以定位和寻址的最小对象是字符。有时程序可能需要一个很小的值，而用一个完整的整数去存储它是一种浪费。通常，我们没有注意节省内存，但当内存紧缺时，我们将会把这些小的值压缩在一起。通常，外部数据格式以及硬件设备接口迫使我们以它为单位进行处理。

例如，磁盘控制器可以操纵内存地址和它们的成员，包括页号（0~15）和页内的偏移地址（0~4095）。算法可能要对页号（4位）、页内偏移地址（12位）以及完整地址（无符号16位）进行操作，能够把页号和偏移地址组合成完整地址，或者从完整地址中抽取出页号和偏移地址。

另一个例子是一个I/O端口，其中的每个位与特殊的条件和操作有关。如果向设备没有阻碍地传送条件，就设置该端口的1号位；如果接收的缓冲区已满，就设置端口的3号位；如果传送缓冲区为空，就设置端口的6号位。算法可能要求单独地设置状态字的每个位和单独地检索状态字的每个位。实现这些计算任务就要求使用位操作和按位逻辑运算。

要把页号和偏移地址组合成完整的内存地址，就要求把内存地址左移12位，并对移位的结果和偏移地址进行按位或运算。

```
unsigned int address, temp;       // they must be unsigned
int page, offset;                 // sign bit is never set to one
temp = page << 12;                // make four bits most senior
address = temp | offset;          // assume no extra bits there
```

把页号和偏移地址从完整的内存地址中抽取出来会更加复杂。为了得到页号，要将地址右移12位，以便把偏移地址的各位去掉，并将页号移到了该字的最低位上。为了得到偏移地

址，要用掩码0x0FFF来进行按位与操作，该掩码的12个最低位均被设置为1：

```
page = address >> 12;          // strip offset bits, get page bits
offset = address & 0x0FFF;     // strip page bits from address
```

为了将某个位设置为1，我们使用三个掩码：每一个掩码只有1位被设置为1，而其他位被设置为0，通过对状态字进行按位或操作：把原来是0的位设置为1，或者让所有原来为1的位保持原来的状态。在前面一节中定义的常量CLEAR、FULL和EMPTY都是只有1位为1而其他位为0的掩码。常量CLEAR的1号位被设置为1，FULL的3号位被设置为1，EMPTY的6号位被设置为1。

```
unsigned status=0;       // assume it is initialized properly
status |= CLEAR;         // set bit 1 to 1 (if it is zero)
status |= FULL;          // set bit 3 to 1 (if it is zero)
status |= EMPTY;         // set bit 6 to 1 (if it is zero)
```

为了重新将某个位设置为0，就需要这样的掩码：除了1位以外其他位均是1。使用按位与操作重新设置状态字中为0的位，而状态字的其他位保持不变。为了重新设置1号位，我们需要一个1号位是0的掩码。为了重新设置3号位，我们需要一个3号位是0的掩码。为了重新设置6号位，我们需要一个其6号位是0而其他位是1的掩码。这些掩码难以表示为十进制甚至十六进制常量。在不同的平台上，我们可能需要不同长度的掩码，这就影响了代码的可移植性。一个常见的方法是将常量取否定来将对应的位设置为0，并在按位与运算中用取否定之后的结果将这些位重新设置为0。

```
status &= ~CLEAR;        // reset bit 1 to 0 (if it is 1)
status &= ~FULL;         // reset bit 3 to 0 (if it is 1)
status &= ~EMPTY;        // reset bit 6 to 0 (if it is 1)
```

为了存取某个位上的值，我们可以使用按位与运算和掩码，这时，除了要存取的那个位外，掩码的其他位均被设置为0。如果这个位的状态为1，则运算的结果就不是0（真）。如果这个位的状态为0，则运算的结果就是0（假）。用于这些运算中的掩码与前面用来设置和重置状态位的掩码完全一样：

```
int clear, full, empty;        // to test for True or False
clear = status & CLEAR;        // True if bit 1 is set to one
full = status & FULL;          // True if bit 3 is set to one
empty = status & EMPTY;        // True if bit 6 is set to one
```

这些对一连串的位（比如寻址）或个别位（比如状态）进行压缩和解压的低层次运算是很复杂的，它们很不直观且易于出错。C++允许我们对不同大小的位域命名，这里使用传统的结构定义来实现。对于每一个域，它们所分配的位数（域宽度）是通过使用一个位于冒号之后的非负常量来表示的。

```
struct Address {
  int page : 4;
  int offset : 12; } ;        // it is not large enough for 12 bits
```

域成员被压缩成机器整数。我们要对有符号整数十分小心：符号通常被分配1位。如果想把所有位都分配给域，该域就必须是无符号的，就像以下的例子那样。

```
struct Address {
  unsigned int page : 4;
  unsigned int offset : 12; } ;        // place for 12 bits
```

位域不能超过一个字的界限。如果一个机器字不能容纳该位域，就会分配到下一个字并且该字的剩余位是空闲的。如果域宽度超过给定平台上的基本类型（它们对于不同机器可以是不同的）的宽度，就是一个语法错误。

域可以节省数据空间，因此没有必要去为每个数值分配一个字节或一个字。然而，由于需要抽取其中的某些位，对这些数值进行运算的代码的规模就会变大。并且最终的结果是不清晰的。

定义这些变量的方式和定义结构变量的方式一样。存取位域也和存取一般结构域的方式一样：

```
Address a; unsigned address;          // make sure that a is initialized
address = (a.page << 12) | a.offset;
```

如果想为某个标记分配1个位，就要保证该域是无符号的。各个域不是非要有名字不可，没有名字的域用于填充。（我们仍然必须指定类型、冒号和宽度。）

```
struct Status {
  unsigned : 1;                       // bit 0
  unsigned Clear : 1;                 // bit 1
  unsigned : 1;                       // bit 2
  unsigned Full : 1;                  // bit 3
  unsigned : 2;                       // bits 4 and 5
  unsigned Empty : 1; } ;             // bit 6
```

处理这个状态变量的代码非常简单。它通过使用类似于我们在本节开头所讨论的例子中的移位和按位逻辑运算来实现。

```
Status stat;                                        // make sure it is initialized
int clear, full, empty;                             // for testing for True or False
stat.Clear = stat.Full = stat.Empty = 1;            // set bit to one
stat.Clear = stat.Full = stat.Empty = 0;            // reset bits to zero
clear = stat.Clear;                                 // the values can be tested
full = stat.Full;
empty = stat.Empty;
```

允许使用零宽度，也允许混合不同整数类型的数据。从一种类型转换到另一种类型的结果会在字边界上分配下一个域。不仔细地使用位域也可能不会减少所占用的空间，正如下一个（设法做到）例子一样。（这段代码是为一台16位机编写的，其中整数被分配了两个字节。）

```
struct Waste {
  long first : 2 ;              // this allocates all 4 bytes
  unsigned second : 2;          // this adds two more
  char third : 1;               // short starts on even address
  short fourth : 1; } ;         // and this: 10 bytes total
```

在某些计算机上，域是从左向右赋值的；而在其他一些计算机上，域是从右向左赋值的。

对于内部已定义的数据结构来说，这并不是什么问题；然而，对于要输入输出的数据来说，这就不同了，比如利用I/O设备缓冲区，当外部数据以某种格式输入而计算机却使用另一种格式时，保存到位域中的数据就可能不正确。

在决定使用位域之前，请考虑一下是否还有其他的选择。请记住，存取一个字符或整数总是比存取一个位域要快，并且编写的代码更少。

5.4 小结

在本章中，我们讨论了程序员在创建大型而复杂的程序时所使用的主要程序设计工具。这些工具大多数是将数据组合成更大的单元：同种类的聚集（数组）和不同种类的聚集（结构）。这些聚集数据类型除了结构的赋值运算之外，没有定义它们自己的运算。所有在聚集对象上的运算都必须以单个元素为单位进行。

由于结构域是通过使用单个域名来存取的，因此它们是相当安全的。数组元素是通过使用下标来存取的，并且C++既不提供编译时保护，也不提供运行时保护来防止下标的非法取值。这很容易导致不正确的运行结果或内存破坏，因此C++程序员应该关心这一问题，特别是对于使用0终止符表示有效数据结束的字符数组情形。

我们也讨论了诸如联合、枚举和位域等的程序员定义的类型。不像数组和结构那样，它们在实际中不是必不可少的，可以编写出任何不用到这些结构的程序。然而，它们可以简化源代码，可以把更多设计人员的意图传达给维护人员，还可以使维护人员（和设计人员）的工作变得更加容易。

第6章 内存管理：栈和堆

在第5章，我们已学习了实现程序员定义数据结构的工具。数组和结构是基本的程序设计机制，它们允许设计人员为设计人员本人、同时也为程序维护人员，以简洁和可管理的形式表达关于应用程序的各种复杂的概念。联合类型、枚举类型和位域帮助设计人员以最容易理解的方式来表示程序代码。

前面所有的程序代码例子中所使用的变量，不论是基本的还是程序员定义的，都是有名字的变量。程序员必须在源代码中为变量指定名字及位置。当程序需要为变量分配内存时，根据C++语言的规定，将自动在称为栈（stack）的内存区域里分配和释放，而不需要程序员更深入地参与。可是，程序员也要为这种缺乏灵活的简便付出代价：每个数据项的大小都要在编译时确定。

为了灵活地定义数据结构，C++允许程序员建立动态数组和链表数据结构，这时就要使用指针。当程序需要给这些动态无名字变量分配更多的空间时，要从称为堆（heap）的区域分配存储空间。因为动态变量没有名字，因此我们可以通过指针间接访问它们。我们将以动态内存管理的复杂性来换取数据定义的灵活性。

本章，我们将学习C++管理栈和堆的技术以及有关的基本方法，诸如使用名字作用域、名字扩展、使用指针的动态内存管理。这些技术是有效利用系统资源的关键。然而，对于初学者而言，动态内存管理可能导致系统崩溃、内存破坏和内存泄漏（当系统运行到内存之外时）。有些程序员喜欢动态内存管理所给予的权力和刺激，而其他的程序员却宁愿尽可能少地使用指针。不管个人爱好如何，都要理解C++所支持的名字管理和内存管理的原则。

在讨论动态内存管理之前，我们将介绍名字作用域和存储类别的概念，它们对于理解C++的内存管理问题很重要。在讨论了动态内存管理这个问题之后，将研究存储在外存上的磁盘文件的使用技术。在磁盘文件中存储数据使程序能够处理无限大的数据集。

注意 本章内容很多。它包含了许多重要的概念和实际的编程技术。没有掌握好内存管理和I/O文件的概念和技术，我们就不可能成为一个技术娴熟的C++程序员。然而，我们可以学习C++其他的内容而不必成为这些领域的专家。如果不能理解这些大而复杂的内容，可以进入到下一章，当觉得自己已经准备好去学习更多的内容时，再回到本章。

6.1 作为合作工具的名字作用域

每一个程序员定义的名字或标识符，在C++程序中都有自己的词法作用域（通常只称为作用域）。

称之为词法的原因是该名字只可以在某一段代码中使用，称之为作用域的原因是在这段代码以外，该名字是不可知的或者它表示另外一个实体。其名字有作用域的实体包括程序员定义的数据类型、函数、参数变量和标号等，在其作用域中的名字可应用于定义、表达式和函数调用中。

6.1.1 C++词法作用域

词法作用域（Lexical Scope）是名字的静态特性。这意味着作用域是在编译时由程序的词法结构确定的，而不是运行时的程序行为。在C++中有6种[⊖]作用域：

- 块作用域。
- 函数作用域。
- 文件作用域。
- 整个程序作用域。
- 类作用域。
- 名字空间作用域。

本章，我们将讨论前面四种作用域。在更详细地解释了类型和名字空间的概念之后，将在后面的章节中讨论另外两个作用域。块作用域也是由开闭花括号定界的，块作用域和函数作用域的区别是函数有参数（以及在该作用域中它们的名字是可知的）和名字。在程序执行期间调用函数时，就进入了该函数作用域，块作用域是不能调用的。在执行了位于其前面的那条语句（如果有的话）之后，块才执行。例如，在每一次通过以下的for循环时，就进入其位于花括号里面的一个无名字块作用域。当函数getBalance()被调用（使用其名字）时，就进入该函数的作用域。（我们将在程序6-1中看到这个函数的实现。）

```
for (i = 0; i < count; i++)
{ total += getBalance(a[i]); }              // accumulate total
```

文件作用域是由该文件的物理边界来界定的。它可以含有类型定义、变量的定义和声明以及函数的定义和声明。在前面的章节中给出的每一个程序都是由文件边界界定的文件作用域。

程序作用域没有界定符号。任何属于该程序的源文件，都位于该程序作用域之中。

6.1.2 同一作用域中的名字冲突

在C++中不允许在一个作用域内有名字冲突。一个名字在其声明所在的作用域中应该是惟一的。在C中，程序员定义的类型可以组成一个独立的空间，这意味着如果一个名字定义为属于某个类型，它就可以在该类型的作用域中使用。编译程序（和维护人员）可以从上下文中弄清楚该名字是类型还是变量。

C++更加严格，所有程序员定义的名字都有一个单一的名字空间。如果一个名字在一个作用域中声明，则在同一个作用域中在所有的名字声明中它应该是惟一的。例如，如果count是一个变量名，则在声明变量count的那个作用域中，其他的类型、函数、参数或另外一个变量都不可以命名为count。

类似于大多数设计语言的软件工程思想，这种做法的目的是为了提高程序的可读性，而不是为了更容易编写程序。当设计人员（或维护人员）在源代码中发现了名字count时，没有必要去搞清楚这个名字是否还有其他的含义：在它的作用域中它只有一个含义。如果设计人员（或维护人员）想在一个作用域中加入名为count的变量，就必须弄清楚这个名字是否已经存在于该作用域中。

这个规则的惟一例外是标号的名字。它们不会和同一个作用域中所声明的或可知的变量、参数或类型等名字发生冲突。由于在C++程序中不会频繁地使用标号，因此这不会导致可读

⊖ 英文原书误写为4种。——译者注

性的降低。可是仍然不要太多地在使用中依赖这种特殊的规则。

与这种惟一性规则相反的是同一个名字可以用于不同的作用域中而不会产生冲突。这个规则减少了设计人员之间所需进行的协调次数。不同的程序员可以在程序不同的部分（不同文件）独立地工作以及命名，而不需要与其他的小组成员进行协调，即使对于同一个文件，如果要协调在不同作用域中所定义的名字，将使设计人员（和维护人员）的工作更加困难。

不同程序实体（数据类型、函数、参数、变量和标号）的词法作用域是不同的。类型名字可以在块、函数或文件之中声明，从它们所在的块、函数或文件的定义位置开始，直到该作用域结束这一范围内，它们都是可知的，而在作用域之外它们是不可知的。对于变量名也一样，它们可以在块、函数或文件之中声明，从其定义的地方开始直到其作用域结束，它们都是可知的。

参数只能定义在函数中。从所在函数的开花括号到闭花括号的范围内，它们都是可知的。标号可以定义在一个块或一个函数中，它们的名字在使用它们的整个函数中都是可知的，而在使用它们的函数之外是不可知的。

C++函数可以定义在一个文件中，但不能在一个块或另一个函数中定义。函数名有着程序作用域，也就是在程序中该函数名应该是惟一的。这个具有全程作用域的名字常常使得开发小组成员之间的协调很困难。如果要在维护期间扩展一个现存的程序也有这种情况：增加新的函数名可能会引起冲突。另一个关于函数命名的麻烦出现在将来自不同厂家（或来自过去的工程）的几个库集成起来时。通常，刚开始该问题并不会出现，直到由不同程序员单独开发的文件在开发周期的晚些时候要连接在一起的时候，它才会出现。

程序6-1给出了一个简单的例子，它输入账户数据，显示数据并计算账户的余额总数。为了简化起见，不是从键盘、外部文件或数据库输入数据（不久我们将那样做），而是使用两个数组num[]和amounts[]来提供账户号码和账户余额的值。当账户号码等于标志（-1）时才停止while循环的数据输入；然后，通过第二个循环打印账户号码，通过第三个循环打印账户余额，并用第四个循环来计算账户余额的总数。这里使用了两种程序员定义的类型：结构Account和整数别名Index，以及函数getBalance()；举这个例子的目的是为了说明作用域的相互影响。为了简单起见，将数组的大小设置得很小。程序的输出如图6-1所示。

程序6-1　类型、参数和变量的词法作用域例子

```
#include <iostream>
using namespace std;

struct Account {                    // global type definition
 long num;
 double bal; } ;

double getBalance(Account a)
{ double total = a.bal;             // total in independent scopes
 return total; }                    // return a.bal; is better
int main()
{
 typedef int Index;                                  // local type definition
 Index const MAX = 5;
 Index i, count = 0;                                 // integers in disguise
 Account a[MAX]; double total = 0;                   // data set, its total
 while (true)                                        // break on the sentinel
```

```
  { long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
    double amounts[MAX] = { 1200, 1500, 1800 } ;      // data to load
    if (num[count] == -1) break;                      // sentinel is found
    a[count].num = num[count];                        // loading data
    a[count].bal = amounts[count];
    count++; }
    cout << " Data is loaded\n\n";
  for (i = 0; i < count; i++)
   { long temp = a[i].num;                            // temp in independent scopes
     cout << temp << endl; }                          // display account numbers
  for (i = 0; i < count; i++)
   { double temp = a[i].bal;                          // temp in independent scopes
     cout << temp << endl; }                          // display account balances
  for (i = 0; i < count; i++)
  { total += getBalance(a[i]); }                      // accumulate total for balances
  cout << endl << "Total of balances $" << total << endl;
  return 0;
}
```

```
 Data is loaded

800123456
800123123
800123333
1200
1500
1800

Total of balances $4500
```

图6-1 程序6-1的程序输出结果

注意 这个程序是由最新版本的32位编译程序编译的，因此没有必要去指明800123456和其他long类型的值。这个程序不能用旧的16位编译程序编译。在第5章相似的程序代码例子中，对这些值加上了后缀L（800123456L等等）；因此它们可由任何编译程序编译。C++程序员应该经常考虑可移植性问题，否则可能会引起错误。查找和更正这些错误是很麻烦的，并且代价是很高昂的。

在这里，类型Account的作用域是整个文件，从其定义的地方开始直到该源文件的结束它都是可知的。类型Account的变量可以定义在这个作用域中的任何地方。在这个作用域中，不管用名字Account做什么事情，比如作为一个整数的名字，都是不正确的。

```
int Account = 5;                  // incorrect use of the name Account
```

类型Index有着函数作用域，从其定义的地方直到main()函数的闭花括号为止它都是可知的。类型Index的变量可以定义在main()中，但不能定义在另一个作用域中，比如在函数getBalance()中。

```
double getBalance(Account a)
{ Index z;                  // syntax error: name Index is unknown here
 return a.bal; }
```

函数getBalance()有着程序作用域。在该程序作用域中，其他对象都不能命名为getBalance。

变量名的词法作用域是最不相同的。C++的变量可以定义为：

- 块变量：定义在块的开花括号后（或在块的中间），从定义的地方开始直到块的结束处是都可见的。在程序6-1中，块变量是在main()中的第一个循环中定义的数组amounts[]和num[]，在main()中的第二个循环中定义的变量temp和在main()中的第三个循环中定义的变量temp。
- 函数变量：类似于块变量，但它们的作用域是一个声明了的函数而不是一个无名块。它们定义在函数体中（在开花括号后，或在中间），从定义的地方开始直到该函数的闭花括号为止它都是可见的。在程序6-1中，函数变量是在main()中定义的i、count、MAX、a[]和total，以及在函数getBalance()中定义的变量total。
- 函数形式参数：定义在函数头部且在整个函数体中可见。这意味着该参数名字可能与在这个函数中定义的变量发生冲突。在程序6-1里的函数getBalance()中只有一个形式参数a。
- 全局变量：有着文件作用域——它们定义在文件中以及位于任何函数之外，且从其定义的地方开始直到文件结束都是有效的。在程序6-1中没有全局变量，作者将在下一个例子中讨论全局变量。

结构的域名对于结构定义对应的块来说是局部的。这意味着可以在这个作用域之外引用它们（不需要进一步的标识符）。在程序6-1中，域名字num和bal只在结构Account中是可知的。因此，bal=0;在main()中是不正确的，因为bal在main()里是不可知的。另一方面，在类型Account变量作用域的任何地方都可以引用这些域（使用选择运算符）。在程序6-1中，它是函数main()的作用域（类型Account的数组a[]定义在其中）和函数getBalance()的作用域（一个Account类型的参数a定义在其中）。由于C++允许程序员在一个作用域中的任何地方定义变量，很重要的一件事是确保该名字在作用域中要先定义后使用。在程序6-1中，在函数main()里常量MAX应该按照词法位于数组a[]、amounts[]及num[]等的定义的前面。

6.1.3 在独立的作用域中使用相同的名字

定义在不同作用域中的名字不会互相发生冲突（有某些例外）。

这里的“不同”实际上需要做进一步的解释。这些不同的作用域应该如何相互关联而使得同一个名字可以在不同的作用域中用于不同的目的？

两个作用域不相交（没有共同的语句）的块是不同的。而且，它们是各自独立的。例如，在文件或在函数作用域中互相跟随（直接或间接）的无名块是独立的，而且可以为完全不同的目的而定义并使用相同的名字。在独立的作用域中定义的名字不会互相冲突。

在程序6-1中，名字temp用在函数main()的两个循环中。实际上，没有必要在这些作用域中使用局部变量：数组元素的域可以直接显示。然而，使用这些变量可以很好地说明作用域的概念。由于这些循环都有着自己的一对作用域花括号，因此名字temp的这些使用所指的是不同的变量，不会互相冲突，并且不要求对它们在使用上进行协调。

对于使用相同名字来定义变量或参数的函数块，也有同样的道理。例如，变量total在getBalance()和main()中都有定义。同样，函数getBalance()不使用局部变量也可以工作，但它的使用可以说明作用域的概念。

类似地，名字a在函数getBalance()中用作一个参数，而在函数main()中用作一个数组。同样，当这些名字定义在独立的作用域时，每一个名字只在它自己的作用域中是可知的，且不需要协调它们的使用。

6.1.4 在嵌套的作用域中使用相同的名字

另外一种不同作用域的类型与嵌套的概念有关。C++是一种块结构语言。这意味着它的作用域可以从词法上相互嵌套，那就是一个作用域的括号全部置于另一个作用域的括号之中。注意，不同的作用域可以是独立的（一个作用域在另一个作用域开始之前结束）或嵌套的（一个作用域在另一个作用域之中），但它们不能交叉。

大多数C++程序使用嵌套作用域。一个无名块可以嵌套在另一个无名块或者一个函数之中。一个无名块不能直接嵌套于文件作用域中，因为控制将不能到达它：它需要函数头。一个函数只可以嵌套于文件作用域中，它不能嵌套于另一个函数中。例如，在下面的设计中，把函数getBalance()隐藏于main()中，以使其名字的作用域将不在文件作用域里，因此当getBalance名字有其他的使用时将不会引起冲突。不可以像下面的程序那样，把一个函数全部嵌套在函数main()中，在C++中这个设计是非法的：

```
int main()
{ double getBalance(Account a)              // idea is illegal in C++
  { double total = a.bal;
    return total; }

  . . . .

  for (i = 0; i < count; i++)
  { total += getBalance(a[i]); }            // accumulate total
  cout << endl << "Total of balances $" << total << endl;
  return 0; }
```

在程序6-1中，循环体作为无名块被实现，它们嵌套在函数main()中；函数main()和getBalance()的作用域嵌套在源文件作用域中。从某种意义上说，文件作用域嵌套在程序作用域中。

嵌套作用域的引入不会改变在外部作用域中所定义的变量或类型的可读性规则。它们在嵌套作用域中是可见的。例如，变量count从其定义的地方起直到函数main()的结束都是可知的，不管函数main()中是否有任何嵌套作用域。因此，当在main()的第一个循环的无名嵌套块中引用变量count时，被引用的是在外部作用域中定义的那个变量。类似地，在所有三个循环的嵌套块中所引用的都是数组a[]的元素。变量定义在main()中并在第三个循环的嵌套块中被引用。

另一方面，定义在嵌套作用域中的变量不能在外部作用域中引用。例如，数组num[]和amount[]定义在main()的第一个循环的块中。因此在那个块的外部不能为main()所使用。如果把程序6-1中的第二个循环用下面的方式改写，在外部作用域中引用num[]，将是不正确的。

```
for (i = 0; i < count; i++)
  cout << num[i] << endl;                   // num[] is not known
```

C++允许在一个嵌套作用域中重新定义已在外层作用域中定义过的实体。在这种情况下，在外层作用域定义的实体在嵌套作用域中变得不可用。当该名字用在嵌套作用域时，它指的

是在这个嵌套作用域中定义的实体。在这个嵌套作用域之外，这个名字所指的将仍是在外层作用域中定义的实体（变量、类型或参数）。

为了展示嵌套的作用，让我们考察一下程序6-2，它给出了程序6-1中程序的一个修改版本。这里删除了无用的代码，在main()的循环体中的局部变量temp和函数getBalance()。此外，变量MAX（实际上它是一个常量）、count和Account的数组a[]在文件里的作用域变为全局，增加了函数printAccounts()：它打印在数组a[]中每一个账户的账户号码和账户余额（显示在单独的一行上）。下标定义在main()的循环中，而不是定义在main()中。程序显示总的余额，然后搜寻一个特定的账户号码并在找到时显示余额。这个版本的输出结果如图6-2所示。

程序6-2　嵌套作用域和名字重定义的例子

```
#include <iostream>
using namespace std;

struct Account {
  long num;
  double bal; } ;

const int MAX = 5;                                  // maximum size of the data set
int count = 0;                                      // number of elements in data set
Account a[MAX];                                     // global data to be processed

void printAccounts()
{ for (int i = 0; i < count; i++)                   // global count
  { double count = a[i].bal;                        // local count
    cout << a[i].num << " " << count << endl; } }

int main()
{
  typedef int Index;
  long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
  long number = 800123123; double total = 0;        // outer scope
  while (true)                                      // break it in the sentinel
  { double amounts[MAX] = { 1200, 1500, 1800 } ;    // data to load
    if (num[count] == -1) break;                    // sentinel is found
  double number = amounts[count];                   // number hides outer number
  a[count].num = num[count];                        // loading data
  a[count].bal = number;
  count++; }
  cout << " Data is loaded\n\n";
  printAccounts();
  for (Index i = 0; i < count; i++)                 // global count
    { double count = a[i].bal;
      total += count;                               // local count
      if (i == ::count - 1)                         // global count
        cout << "Total of balances $" << total << endl; }
  for (Index j = 0; j < count; j++)
    if (a[j].num == number)                         // outer number, global array
  cout <<"Account "<< number <<" has: $" << a[j].bal << endl;
  return 0;
}
```

全局变量的作用域是它们的定义所位于的文件。在该文件中的任何函数都可以引用这个名字（除非该名字被隐藏），且所有的引用都将指向同一个全局变量。例如，在程序6-2中的数组a[]和变量count[]只在函数printAccounts()和main()中引用，常量MAX只

用在main()中。没有必要为了使用它们而在printAccounts()和main()中定义这些名字。有全局定义已经足够。

```
  Data is loaded

800123456    1200
800123123    1500
800123333    1800
Total of balances $4500
Account 800123123 has: $1500
```

图6-2 程序6-2中程序的输出结果

从某种意义上说，全局变量的作用域是程序作用域而不是文件作用域。如果我们在同一个程序的另一个文件中将名字MAX、count或num定义为全局名字，编译程序将单独地编译每一个文件，因为在编译期间编译程序不会检查其他文件的内容。然而，连接程序将报告定义的副本，不管这些名字是用于相同的或完全不同的目的。例如，a[]和num[]可以在另一个文件中定义为纯量类型的变量而不是数组，那么这个副本的使用就是一个错误。这时只有全局定义才是正确的，它既不适用于声明也不适用于非全局定义。我们很快将看到这样的例子。

其他的C++作用域（函数或块作用域）定义在一个特殊的嵌套于全局文件作用域中的源文件中。因此，全局名字在该文件的函数中可见，正如任何外部名字在嵌套作用域中可见一样。如果函数自己有嵌套作用域，全局变量的名字在这些嵌套作用域中仍然可见。在程序6-2中，全局数组a[]和num[]及下标count都用于嵌套在main()作用域中的第一个循环体中。同样，嵌套作用域（任意深度）的存在不改变在外层作用域中定义的名字的可见性。

嵌套作用域可以使用在外层作用域中定义过的名字来定义变量（因此在嵌套作用域中已可知）。当这个名字用在局部嵌套作用域（文件的一个函数、函数中的一个块或另一个块）中时，这个引用是局部名字的定义。当这个名字用在外层作用域中时，这个引用是在外层作用域中的定义（因为在作用域之外局部名字不可知）。

在程序6-2中，函数printAccounts()在循环的测试条件中使用名字count。这个名字指的是全局变量count。然而，在循环中名字count指的是在循环体中定义的变量，而不是全局作用域中的定义。嵌套名字重定义了全局名字。注意，嵌套名字不必定义一个与原来的类型相同的变量，它可以定义为任何其他类型。

不使用变量count来编写函数printAccounts()并不困难，作者引入它的目的只是想以一个相对简单的例子来说明名字作用域的概念。实际上，不可能给出一个必须重用某个全局名字才能完成要求的例子，我们总是可以利用一个不同于外层作用域中的局部名字。名字作用域的概念使我们不必去另外设计一个不同的名字，而可以使用一个自己喜欢的名字重新定义外部名字。

当嵌套作用域重定义了外部作用域（这里的外部作用域指的是全局作用域，或者嵌套作用域中的作用域）中的名字时，外部作用域中的名字在嵌套作用域中是不可见的。重定义外部作用域中的名字可以告诉维护人员：设计人员不希望在局部作用域中使用全局名字。

在程序6-2中，main()中的第一个循环体定义了变量number，而main()本身的作用域定义了同一个名字。这就意味着，当循环体使用number时，它指的是类型为double的局

部变量而不是类型为int的外部变量，因为嵌套的名字重定义了外部名字。然而，在被嵌套循环体外，例如程序6-2倒数第二行中的number，则表示main()中定义的变量。

类似地，程序6-2中main()的第二个循环的循环体定义了double类型的变量count，它重新定义了整数类型的全局变量count。在该循环中所引用的名字count会由编译程序解释为对double类型的局部变量的引用，尽管在循环的测试条件中它指的是int类型的全局变量count。

如果在嵌套作用域中也希望存取全局名字，它可以使用C++全局作用域运算符'::'来存取全局名字。例如在程序6-2中，余额总数是在第二个循环里而不是在循环之后打印出来的（这将会更加简单和自然）。因此，该循环必须将下标i与数据集合中的有效元素个数进行比较。这时，在main()的第二个循环中所用的::count是指全局对象count而不是指局部对象count。

被隐藏的全局对象不应该随便存取。如果在嵌套作用域中需要该全局名字，该全局名字就不应该被重定义。因为，嵌套作用域可以自由地使用任何名字去避免名字冲突。然而，在维护过程中若有新的要求需要使用已重定义的全局名字时，就可能需要使用这个全局作用域运算符，因为在原来的设计中没有预期到这个需要。

警告　全局作用域运算符::不考虑作用域规则。对于维护人员来说，假定作用域规则不变比去搜寻全局作用域运算符的作用域更为容易。对变量的名字要尽可能少地使用该作用域运算符。

注意，全局作用域只存取全局变量。C++没有为嵌套作用域提供方法以便从外层作用域中存取已被嵌套作用域重定义的变量。

在程序6-2中，第一个循环的循环体定义变量number，它隐藏了在main()中定义的变量number。这意味着在第一个循环体中所有对number的引用都是局部变量。在main()中定义的变量number只能在这个循环的循环体之外存取（比如在程序6-12中的最后一个循环）。

注意　全局作用域运算符存取全局名字。如果一个嵌套作用域重定义已在外层块中定义的名字，嵌套作用域就丧失了引用外层定义的名字的能力。如果嵌套块需要那个外部名字，就不要在嵌套块中重新定义它。

6.1.5　循环变量的作用域

在循环头部定义循环变量是模仿Ada的一种技术，但C++在实现它时会有所不同，而且不同的编译程序的实现也有所不同。如果循环变量与在外层作用域中定义的名字相同，有些编译程序会把它标识为错误，而有的却不会。当循环变量用于循环体之外时，有些编译程序会标识它为错误，而有的却不会。新的C++标准把循环变量的作用域限制在循环体内。因此，它不该在循环之外使用。当位于同一个作用域中的一个循环使用了相同的名字作为另一个循环的变量时，有些编译程序会标识它为错误，而有些却不会，尽管新标准允许那样做。程序6-2给出了使用这个技术的一个深思熟虑的小例子：循环变量没有重定义已在外层作用域中定义的名字，它们不在循环体之外使用，以及不在位于同一个作用域中的其他循环中重定义。

一般说来，词法作用域是一种重要的机制：名字可以重用于独立的作用域中（没有冲突）并在嵌套的作用域中重定义（隐藏外层名字）；当具有相同名字的作用域对象嵌套时，最近

定义的名字隐藏次近定义的名字。

作用域规则帮助我们避免名字冲突和程序员之间过多的协调。

6.2 内存管理：存储类别

在上一节中讨论的词法作用域是程序的一种编译时间特性，它决定了某个特定名字在某段程序源代码中是可知的。然而，它没有决定在执行期间什么时候为一个特定变量分配内存，以及什么时候收回这个内存以供其他变量使用。在运行时内存分配的规则依赖于程序员定义名字的另一个特性：它们的存储类别（或者是范围）。

存储类别指的是当变量名和它在内存中单元之间的关联是合法时（即内存空间分配给该变量时）的一段有效的执行时距。不像词法作用域，存储类别是程序行为的运行时特点。

在C++中，程序的执行总是从main()开始的；在main()中第一条可执行语句通常是程序执行的第一条语句。函数main()调用程序的其他函数，而这些函数又调用另外的函数。当一个服务器函数完成其执行（它执行了一条return语句，或它执行到函数体的闭花括号）后，控制就返回到调用它的客户函数。当main()调用的最后一个函数终止且main()的执行达到闭花括号（或一条return语句）后，该程序就终止。

到目前为止，我们见过了两种版本的main()函数，一种具有int返回值类型和另一种返回值为空值类型。当返回值类型为空值时，编译程序就会假定返回值是整数类型（当然，这不是所期望的）。每种形式的main()都可以有可选的参数。

```
void main(int argc, char* argv[])       // command line arguments
{ for (int i = 0; i < argc; i++)        // start of program execution
   cout << "Argument " << i << ": " << argv[i] << endl;
. . . . . }                             // end of program execution
```

当main()执行时，其参数是从操作系统传递给main()的。它们包括在程序调用期间用户要显示的命令行参数信息（如果有）。这些参数被定义为命令行参数的数目（argc）和字符串数组（向量）(agrv[])，其中每一个字符串包括一个命令行参数。（随后我们将讨论数组的指针符号。）

通常，这些字符串是写在命令行上的文件名。在上面的例子中，函数main()使用命令行参数的数目来检查每个参数。这种情况下它只是显示每一个参数。程序的可执行文件的名字包括在命令行参数的列表中。在字符串数组中，它的下标从0开始。例如，如果可执行文件的名字是copy，那么命令行

```
c:\>copy account.cpp c:\data
```

将打印下列行：

```
Argument 0: copy
Argument 1: account.cpp
Argument 2: c:\data
```

在程序执行过程中，程序变量（对象）可以分配在为程序预留的三个内存区域中：固定内存、栈内存和堆内存。在这个讨论中，了解具体的计算机怎样管理这些内存区域并不重要。不管它是一个标量类型的变量、一个数组、一个结构或数组类型的变量，还是一个联合或枚举类型的变量，在程序执行期间都将根据它的存储类别将它分配在这些内存区域中的某一个区域中。

存储类别的概念进一步改善了名字作用域的概念。在文件作用域中定义为全局的变量存放在固定区域中。定义为一个函数或块的局部变量存放在栈中。另外，C++支持动态变量。它们没有定义为全局或局部变量，因此它们没有名字。因此，它们由显式的程序语句来分配内存空间（运算符new）。动态变量分配在程序的堆中。

在变量的定义中，通过使用下列的关键字可以确定C++的存储类别。

- auto：是在一个块作用域或函数作用域中定义为局部变量的缺省类别（自动变量）。
- extern：适用于具有文件作用域的全局变量。
- static：可以作为文件作用域中的全局变量，也可以作为一个块中或函数中的局部变量。
- register：用于指定存放在高速寄存器中而不是存放在随机存取内存中的变量。

对于这些存储类别的对象（变量），语言的规则定义了其分配和释放的情况：extern和static变量分配在程序的固定数据内存中，auto变量分配在程序的栈中，而register变量分配在寄存器中——如果可能的话。如果没有足够的寄存器可以利用，这些变量要么分配在固定区域中（对于全局变量），要么分配在程序的栈中（对于局部变量）。

6.2.1 自动变量

自动变量是定义在函数或块中的局部变量。修饰符auto是缺省的，并且不常使用。例如在程序6-2中的函数printAccounts()也可以写成这样：

```
void printAccounts()
{ for (auto int i = 0; i < count; i++)          // global count
  { auto double count = getBalance(a[i]);       // local count
    cout << a[i].num << " " << count << endl; } }
```

由于C++程序员不喜欢额外的输入，如果没有好的理由一定要这样做，他们更喜欢省略这些缺省的修饰符。

当程序的执行进入函数或块的开花括号时，就从栈中分配自动变量的存储空间。如果定义中包括了初始化操作（如前面printAccounts()的例子），则会同时初始化为该变量分配的存储空间。如果在定义中没有指定初始值，则该变量的值就是不确定的。它很可能是一个上次分配给该变量的内存单元所遗留下来的值。无论如何，对于想弄清楚该不确定的值是什么以便在程序中使用它，是不太可能的。“undefined”这个词虽不是C++关键字，但我们应该非常认真地对待它。如果我们需要使用一个具体的值，那就初始化该变量并使用它，千万不要依赖于不确定的值。它们可能是任何值，并且它们在不同的程序执行中可能是不同的，即使凭经验你认为它们是相同的。请不要过于相信你的经验。

在程序执行过程中，自动对象只有当控制进入它们的定义所在的作用域中时才存在于内存中。否则自动对象分配在程序的栈中（且根据名字引用），直到执行到达该作用域的闭花括号为止。此时，它们的内存被还给栈，还可以用于其他目的。

这是内存管理的一项伟大技术：它将程序员从为单独的计算目标分配和释放内存的责任中解放出来。对于某些任务，这个技术是不够完善的，这时就要使用动态内存管理来代替。从本章后面的内容能看到，动态内存管理更加复杂和容易出错。因此应该尽可能多地使用自动变量（实际上也是这样）。

在对同一个函数的再次调用（或者同一个循环的再重复）中，为同一个自动变量分配的

内存可能不在同一个栈单元并具有相同的内容。因此，自动变量不能在函数的连续调用之间或循环的连续重复之间传递数据。如果变量没有初始化，那么它在每次分配中都有着不确定的值。如果变量的定义包括了初始化，那么每次进入该作用域时就重复这个初始化。在printAccounts()这个例子中，局部变量count在通过该循环的每次重复中都会分配内存、初始化以及释放内存。变量i的存储为每次对printAccounts()的调用分配内存、初始化以及释放内存。

当程序有足够的内存和执行速度时，我们不需要为局部变量优化内存管理。当资源紧缺时，优化设计就变得很重要。例如，在程序6-2中将数组num[]定义为函数main()中的一个局部变量，并将数组amounts[]定义为第一个循环体中的一个局部变量。这两个数组都含有从全局数组a[]中输入值的一些数据。在程序的不同地方定义数组num[]和accounts[]就把本该放在一起的东西分开了。

这个做法也可能会牵涉到其性能。数组num[]只在函数main()开始执行时分配空间。数组amounts[]则重复地进行分配、初始化和释放，其次数和循环体的执行次数一样多。数组的分配和释放不会占很多执行时间（它包括计算栈指针），但为了初始化而把值拷贝到数组元素中所占的时间和把数组amounts[]中的数据拷贝到数组a[]中所用的时间一样多。在同一个地方分配数组num[]和amounts[]将是较好的，它只需要在程序执行期间进行一次。

```
int main()
{ typedef int Index;
  long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
  double amounts[MAX] = { 1200, 1500, 1800 } ;          // data to load
  long number = 800123123; double total = 0;           // outer scope
  while (true)
  { if (num[count] == -1) break;
    . . . . . } }                                      // end of main()
```

自动变量在它们的作用域之外是不可见的，因此它们可以在其他作用域中重用，在不同作用域中的名字的内存单元之间没有什么联系。从减少开发者之间协商的观点来看，这是很好的。当一个全局变量在不同的作用域中使用时，在每一个作用域中所访问的都是同一单元。因此必须研究在每个函数中全局变量的使用，以便弄清该单元是否真的可以多次使用，或者是否要引入不同的变量。自动变量的使用简化了设计人员以及维护人员的工作。

根据作用域规则，一个名字可以在一个嵌套块中重用为另一个对象。具有相同名字的新对象在栈中分配的单元与定义在外层作用域的同名变量的单元是不同的。在嵌套作用域中的名字隐藏了先前已在栈中分配的对象（它仍是有效的）。例如在程序6-2中，变量number定义在函数main()中，以及在main()的第一个循环的循环体中重定义。第二个变量number在每次循环开始时分配在栈中，并在每次循环结束时释放掉。它会分配一个完全不同的单元（实际上，每次循环它都可能不同），且它与在main()的开头为number分配的栈单元没有什么联系。因此，当main()中的第二个循环要使用曾在main()的开头赋给number的值时，这个值还是原封不动的，而number又可以在第一个循环的嵌套作用域结束后继续使用。

类似地，当main()调用printAccounts()时，变量count的内存是为了printAccounts()的每次循环而从栈中分配的。对于每一次循环，分配的单元可能是不同的，并且它们中没有一个和全局变量count在固定区域中的单元有任何关系。

如果嵌套作用域没有隐藏已在外层作用域中定义的变量，则那个变量的名字在嵌套作用域中是可利用的。在程序6-2中，变量total是在main()的开头分配的，且没有在它的嵌套作用域中重定义。当第二个循环在它的循环体中引用total时，它引用的是定义在外层作用域中的变量。

函数的形式参数被当做是定义在该函数作用域中的自动变量；它们利用在函数调用中的实际参数的值来初始化。例如在程序例子中的第一个版本中（在程序6-1中），函数getBalance()利用main()中a[]的值来初始化它的参数。参数的内存是在函数执行开始时在栈中分配的，并且当执行到达函数的闭花括号时释放参数的内存。

一般来说，在块的尽可能深层的嵌套结构中定义变量是一个好方法。这样做可以有以下的优点：

- 它最小化了名字可知的程序作用域，因此最小化了与其他对象的名字冲突的潜在性。
- 可以在最短的时间内为这个变量分配好内存；而在这段时间之外，该内存可以为其他目的所重用。

要权衡的是在程序的其他部分是否容易存取对象，以及由于重复的分配、初始化和释放所引起的对性能的负面影响。另一个权衡是可能存在用完栈空间的危险：总的内存需求依赖于函数调用的顺序，编译程序和程序员都不能准确地预测它。因此当数组在函数和嵌套块中定义为局部变量时，这种权衡就特别重要。例如，在程序6-2中的数组amounts[]。

6.2.2 外部变量

外部或全局变量是那些定义在函数之外的变量。与在6.1节提到的一样，它们的作用域是其声明所在的文件，从其定义的地方起直到文件结束。因此，另一个文件不能通过它来引用相同的变量，因为该名字在另一个文件中是不可见的。（实际上，这可以通过一些努力来做到。）这个名字也不能用于另一个文件中来定义其他外部变量。从这个意义上说，全局变量名有着程序作用域，类似于C++的函数名。

全局变量的内存分配不同于自动变量，其空间分配在固定数据区域中。它是在程序执行开始时分配的，即在main()函数的第一条语句执行之前分配空间。该内存单元直到程序终止之前一直和该变量名有关联，且在main()的最后一条语句执行之后才释放。

全局变量的定义也允许初始化。如果没有初始化，该变量就初始化为某个类型的0值。这是和自动变量一个重要的不同之处。自动变量没有缺省的初始值且初始状态是不确定的（程序员通常称之为垃圾）。

在程序6-2中，变量MAX、count和a[]定义为全局变量。

在程序中所有全局变量需要的内存总数很容易计算出来。编译程序单独地编译每一个文件，并通过累计所有全局对象的大小来计算全局变量要求的空间。（这对于自动变量没有什么意义，因为它们不在同一时间存在于内存中。）使用全局变量的另一个优点是速度。由于每个全局变量只分配和释放一次而不是每次进入作用域都进行一次分配，分配内存的操作不会降低程序的执行速度（当然，对于很多应用这都是不重要的）。

使用全局变量的另一个优点是对程序的栈空间的更少要求。程序要求的栈的大小不可能准确地估计，因此总存在着用完栈空间的可能性。因此若没有很好的理由就不要增加对栈空间的要求。例如，程序6-2中的数组amounts[]被定义为局部的，而数组num[]被定义为

全局的。这样不仅将逻辑上应该在一起的东西分开了，而且在每次循环时对数组amounts[]进行分配和初始化，并把数组amounts[]分配在栈中。前面两个操作要求时间，而第三个操作要求额外的内存。如果把这个数组说明为全局变量就会消除这些缺点。在这个例子中，由于数组只有3个元素，它不会损坏栈。但很多程序员会把大的数组分配在栈中而没有意识到栈的大小问题。

使用全局变量的另一个优点，至少对于某些程序员来说，是有机会避免使用函数参数。由于一个全局变量的作用域是其定义所在的文件，因此，定义在同一个文件中且在那个全局变量定义之后的任何函数代码都可以直接存取该变量。例如，在程序6-2中的函数printAccounts()直接存取全局变量count和a[]，而不用复杂的参数传递。其他程序员认为对全局变量的直接存取将无法向维护人员传达该函数的界面是什么。为了搞清楚该函数使用了哪些变量和改变了哪些变量，就要检查函数的每一行代码。我们不久将看到，使用参数就可以直接表明函数接口，而没有必要去检查每一行代码。

在程序执行的整个时期扩大全局变量的生命周期的负面影响是，在程序中恢复内存给其他变量使用将变得更困难。例如，在程序6-2中变量count和a[]作用于整个程序中。另一方面，数组num[]和amounts[]只需要作用在main()中第一个循环的循环体中。在此循环之后，数组amounts[]的空间可以恢复给其他变量使用。然而，数组num[]的空间一直保持在那里，如果其他的场合要重用它就需要细心的规划，且在维护期间这会变得很困难。因此我们不把所有的程序变量都定义为全局变量。

在源代码文件中定义为全局变量的名字在任何嵌套于该文件中的作用域中都是可知的。我们可以从该文件的任何地方存取全局变量。例如在程序6-2中，count在main()中作为循环的限制，MAX则用来定义数组a[]、num[]和amounts[]。在main()中引用了全局数组num[]，在函数main()和函数printAccounts()中引用了全局数组a[]。

我们以前曾提过，一个嵌套作用域可以重定义（隐藏、重写）全局名字。这个重定义的空间是从栈中而不是从固定区域分配得来的；而且，在这个嵌套作用域中通过这个名字引用的是局部的自动变量，而不是全局变量。在程序6-2中，函数printAccounts()使用的名字count是一个自动变量，在main()中的第二个循环也一样。当作用域运算符'::'和重定义的名字写在一起时，它引用的是在固定数据区域中的内存单元上的变量，而不是栈中的内存单元上的变量（例如在程序6-2中的::count）。

如果程序6-2中另一个文件在它的一个函数中定义了一个局部变量count，将不会引起什么问题，因为这些作用域是独立的。这个函数将引用在栈中的内存单元上的变量。然而，如果另一个文件定义一个全局变量count（这应该是一个表达能力很强的常见的名字），程序将不能连接。全局变量的使用使得开发该程序的不同文件的程序员之间要进行额外的协调。

然而，在一个文件中定义的全局变量可以在应用中被另一个文件所引用。这也是另一个使用全局变量的理由。

关键字extern用来使得在一个文件中定义的全局变量在另一个文件中是可知的。这不是为了重用该全局名字，而是为了使用同一个名字来引用同一个内存地址

如果将程序6-2分割成更多的函数，把这些函数放进不同的文件中，那么更多的程序员就可以参加该程序的开发工作。假定我们不在main()的结束处搜寻一个特定的账户，而是想去调用函数printAverage()，它使用在main()中计算的账户余额总数作为它的参数，

并且打印平均余额。我们不在cout语句中使用数值，而是利用变量caption[]，它含有文本“Average balance is $”（使程序更容易国际化的一种常见技术），且利用函数printAverage()去调用函数printCaption()——它使用了变量caption[]。同样，这里给出小例子以便它们相对容易理解，但我们引入了额外的函数来讨论，这对于大程序的并发是很重要的。

为了在另一个源文件中实现printAverage()和printCaption()，必须确保两件事情：

- 调用函数printAverage()的源文件，即有main()的文件，必须知道printAverage是定义在其他某个文件中的一个函数的名字。
- 实现printAverage()和printCaption()的文件知道在其他某个文件中定义了全局变量count和caption[]。

程序6-3给出了解决这类问题的程序6-2的修改版本，它简化了函数printAccounts()的实现，消除了类型Index，数组amounts[]定义在数组num[]的旁边（这两个数组本来就应该放在一起），函数printAverage()在main()的结束处被调用。它添加了一个全局数组caption[]，其中含有与平均余额一起打印的信息。程序6-4给出了实现printAverage()和printCaption()的第二个文件。程序的输出如图6-3所示。

程序6-3　通过外部声明而与其他文件交流（第一部分）

```
#include <iostream>
using namespace std;

struct Account {                                    // global type definition
  long num;
  double bal; } ;

extern void printAverage(double total);             // defined elsewhere

const int MAX = 5;
Account a[MAX];                                     // global data to be processed
int count = 0;                                      // number of elements in data set
char caption[] = "Average balance is $";            // caption to print

long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
double amounts[MAX] = { 1200, 1500, 1800 } ;        // data set to load

void printAccounts()
{ for (int i = 0; i < count; i++)                   // global count
   cout << a[i].num << " " << a[i].bal << endl; }

int main()
{
  double total = 0;
  while (true)                                      // break on sentinel
  { if (num[count] == -1) break;
    a[count].num = num[count];                      // global a[], num[], amounts[]
    a[count].bal = amounts[count++]; }              // load data
  cout << " Data is loaded\n\n";
  printAccounts();                                  // local function
  cout << "\n Data is processed\n\n";
  for (int i = 0; i < count; i++)
```

```
    { total += a[i].bal; }
  printAverage(total);                    // global in another file
  return 0;
}
```

程序6-4 通过外部声明与其他文件交流（第二部分）

```
#include <iostream>
using namespace std;
extern count;                             // defined and initialized elsewhere
extern char caption[];                    // defined and initialized elsewhere

void printCaption()                       // called from this file only
{ cout << caption; }

void printAverage(double sum)             // called from another file
{ if (count == 0) return;
  printCaption();
  cout << sum/count << endl;
}
```

```
  Data is loaded

800123456    1200
800123123    1500
800123333    1800

  Data is processed

Average balance is $1500
```

图6-3 程序6-3和程序6-4中的程序代码的输出结果

在程序6-3中，通过把以关键字extern为前缀的printAverage()原型加到源文件中，我们解决了第一个问题。

```
extern void printAverage(double);          // it is defined elsewhere
```

如果在函数声明中省略了关键字extern，编译程序和连接程序则一定要搞清楚其函数接口。有些C++程序员更喜欢使用该关键字，以防止可移植性问题的出现。

```
void printAverage(double);            // still, it is defined elsewhere
```

当将关键字extern用于变量时，有两个含义：首先，它表示在这个文件中定义的全局变量在另一个文件中是可见的；第二，它表示一个变量在另一个文件中定义并且在这个文件中声明，以便在这个文件的函数中该变量是可见的。在第一种含义中，extern的使用是可选的；在第二种含义中，extern是必不可少的。

这似乎很复杂，其实不然：extern在定义中是可选的，而在声明中是必不可少的。让我们看看在程序6-3中的外部变量的例子。在程序6-3中的全局变量都有定义。因此，它们是隐式的外部变量：它们在另一个文件中是可见的，且没有必要去使用extern关键字。当使用时，如果没有初始化该变量，它也不会造成什么破坏。

```
extern int count = 0;             // OK: this is a definition
```

初始化的出现告诉编译程序这是一个定义，而不是一个声明。如果省略了初始化，那么没有初始化的定义就变成了一个声明，而且连接程序将count解释为缺少定义。

```
extern int count;                    // this is a declaration
```

同时缺省初始化和关键字extern又使它成为一个定义（当然，该值应该在其他地方初始化），且可以从另一个文件存取该变量（程序6-4定义了printAverage[]）。

```
int count;                           // OK: this is a definition
```

警告　所有全局变量都缺省为外部变量。关键字extern的使用是可选的；用它可向维护人员表明该变量可以用在其他文件中。然而，如果全局变量没有在定义时初始化，连接程序就会把它误认为是一个声明——如果使用了关键字extern。

在程序6-3中数组caption[]被初始化。因此，这是一个定义（在固定区域中为数组分配内存），且该数组缺省为extern，因此可以用在另一个程序6-4中的文件中，其中定义了printCaption()。数组num[]和amounts[]也是全局变量，因此也可以用在其他文件中。但实际上它们没有用在其他文件中（也不应该在其他文件中使用，因为它们包含了程序的初始化数据）。但维护人员不能很明显地从设计中看出来这个事实，即caption[]用于另一文件，而num[]和anounts[]没有用在其他文件中。我们将通过引入静态存储类别来解决这个问题。

程序6-4给出了函数printCaption()，它被这个文件中的printAverage()调用，并且使用了在程序6-3文件中定义的数组caption[]。为了使其成为可能，在程序6-4中数组caption[]被定义为extern。变量count没有初始化，也定义为extern。这使得它成为一个声明。省略了关键字extern将使它成为一个定义，且连接程序将标识count的两个定义有错（即使类型是不同的）。然而，编译程序单独编译各个源文件时会漏掉这个问题。关键字extern的使用允许一个文件去存取在其他文件中定义的数据和函数，但它并没有告诉维护人员哪些全局变量和函数将用于其他文件中，比如像printAverage()那样；以及有哪些不这么用，比如printCaption()。同样，使用关键字static将解决这个问题。

另外要注意数组（caption[]）的声明不要求指定数组的大小，因为声明没有分配内存：它们表示这个对象在其他地方分配内存。类似地，也不应该初始化extern对象，这将使一个声明变为一个定义（以及产生名字冲突）。

与定义不同，外部声明可以在不同的文件中重复，甚至在同一个文件中重复。有了这些声明，文件中的代码就可以使用该全局名字，就好像该变量是在这个文件中定义的一样。例如，在程序6-4中，函数printAverage()引用了count、函数printCaption()引用了caption[]，而count和caption[]是在其他文件（程序6-3）中定义的。

外部变量提供了组成大程序的不同文件的函数之间的一个良好的通信工具。只有在将这些函数分散在不同文件中的好处比将这些函数放在同一个文件中更多时，我们才会使用外部变量。程序6-3和程序6-4给出了文件之间通信的一个例子。把应该放在一起的东西放在一起可以消除文件之间通信的需要，消除extern的需要，简化设计和维护的任务，以及减少可能的错误。

6.2.3　静态变量

关键字static在C++中有五种含义。所有的静态变量都有着一些共同的特点。（它们都

分配在固定内存区域中而不是栈中）。然而，不同含义之间的差别是重要的，而且在不同的环境中使用同一个关键字可能会变得令人费解。下列的C++实体可以定义为static。

- 只能由与变量在同一个文件中定义的（而不是在其他文件中定义的）程序存取的全局变量。
- 在一个函数中（或一个无名块中）定义，并且要求它们的值在调用另一个函数时（或从一个作用域执行到另一个作用域的执行）能够保存下来的局部变量。
- 只能引用同一类型变量的单一内存单元的结构（或类）域。
- 只存取参数、全局变量或类的静态变量而不存取非静态类域的类成员函数。
- 只有在同一个文件中定义的客户代码才能对其进行存取的全局（非成员）函数。

目前，我们还不能完整地探讨所有的问题，在这里只讨论前两种和最后一种情况，另外两种情况将在第8章中讨论。

关键字static的第一种用途是使全局变量成为一个文件的私有变量，以致其他文件不可以通过将它们定义为extern而存取这些变量。例如，程序6-3中定义了全局变量MAX、a[]、count、caption[]、num[]和amourts[]，但它没有指明哪些变量将会被其他文件所存取。为了表示只有count可以在其他文件中存取，而存取其他所有全局变量的函数必须在同一个文件中（并确保没有别的文件可以存取这些全局变量），程序6-3应该将其他全局变量定义为static。

```
int count = 0;                    // it can be made extern elsewhere
static const int MAX=5;           // it cannot be made extern elsewhere
static Account a[MAX];            // no access from code in other files
static long num[MAX]={ 800123456, 800123123, 800123333, -1 } ;
static double amounts[MAX] = { 1200, 1500, 1800 } ;
```

在全局变量定义的前面加上关键字static，既不会改变所分配的内存单元（固定存储区）也不会改变变量的生命周期（从开始到程序结束），惟一的改变是使这个变量不能在其他源文件中被定义为extern，因此在程序的其他文件中不可以存取它们。这个程序设计技术深受程序员的好评。

注意数组caption[]不在这些全局变量之中。由于它只为函数printCaption()所使用（在程序6-4中），它不应该离开这个函数而放进程序6-3中，因为那里没有函数存取它，它应该回到程序6-4中。由于定义在文件中的其他函数都不存取这个数组，它可以（也应该）在程序6-4中声明为static。以下就是程序6-4开始时的定义情况：

```
extern count;                     // defined elsewhere
static char caption[]             // no extern, defined and init here
 = "Average balance is $";        // used locally, not in other files
```

有些程序员相信这样做主要与安全有关。使用这种静态变量可以防止程序其他部分的偶然或不允许的改动。这是事实，但这种错误是很少出现的。更为普通和更加重要的是，这个技术的真正价值是消除了程序员之间的通信，通过将全局变量定义为static，使得其他程序员不需要协调名字的选择，从而使得在程序的任何文件中都可以使用诸如MAX、a、num、amounts以及caption之类的非常具体而且常用的名字。

通常，全局变量的使用应该受到限制。当它们用于实现同一文件中的函数之间的通信时，它们应该是静态的，这样可以减少对开发其他程序文件的程序员的干涉。只有当真正需要在其他文件中存取全局变量时，才将它们定义为非静态（但要检查程序是否把本该放在一起的

东西分开了）。当然，当一个全局变量定义为static时，它就不能在另一个文件中存取了。如果它不是static的（如程序6-3中的count），并没有保证它一定要在其他文件中存取，因为程序员可能会忘记把这个变量只在一个文件中存取的意图传达给维护人员。因此我们必须特别小心使用全局变量。

将全局变量定义为static的技术在C中非常重要。将数据和函数在同一个文件中捆绑在一起（比如把数组移到程序6-4之后的数组caption[]和函数printCaption()），数据被定义为static，因此在外面是不可见的，在一个文件中的函数可能将被在其他文件中的客户函数调用。

在C++中，将数据和函数在类中捆绑在一起，这减弱了使用全局变量的压力。此外，使用名字空间进一步减少了对全局变量的需求。因此，上述技术在C++中的重要性没有在C中那么明显。可是当我们将变量定义为全局时，仍然不要忘记将它们定义为static以便消除对其他设计人员的干扰。

关键字static的第二个含义与第一个是不同的。当对定义在一个函数中或一个块中的局部变量使用static时（记住，这些变量的缺省类型是自动变量），就把该变量从栈移到了内存的固定存储区中。现在，这个内存单元的生命周期不是从定义的开始到函数或块的结束（这是自动变量的），而是从程序的开始直到执行的结束。这意味着在这个单元上的值在该作用域的一次执行中被设置，而在下一次进入该作用域时可以利用。对于该变量名，仍然按照本章第一节中讨论过的作用域规则来使用。在该变量定义所在的花括号之外，其名字是不可知的。因此，其他独立的作用域，即使在同一个文件中，也可以使用同一个名字。而且，在不同作用域中的变量可以使用相同的名字并定义为静态变量，这不会引起名字冲突，即使所有这些变量都分配在固定存储区里。由于它们在不同的作用域中，因此变量的名字在程序执行的不同时刻都是可知的。

例如，在程序6-3中的函数printAccounts()可能改为只打印一个账户。为此，可以定义一个全局变量i并在printAccounts()中将它作为下标。

```
const int MAX = 5;
Account a[MAX];              // global data to be processed
int count = 0;               // number of elements in data set
int i;                       // global index
. . .
void printAccounts()
{ cout << a[i].num << " " << a[i].bal << endl;
  i++; }                     // increment index after use
```

在main ()中，用一个循环调用printAccounts():

```
for (int j = 0; j < count; j++)
  printAccounts();
```

C++语言也允许在循环中使用下标i，但我们当前的编译程序版本不允许这样做。（在程序6-3中的最后一个循环定义i。）这个设计的缺点是使用了更多的全局变量（污染了全局空间）。为了避免出现这个问题，可以把下标的定义移到printAccounts()中，以避免出现与使用这个名字的其他用途之间的潜在的冲突。

```
void printAccounts()
{ int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
```

```
  i++; }                          // increment index after use
```

问题是，这里下标是一个自动变量，因此每次从main()调用printAccounts()时，它都在栈中得到新的空间。因此，它不能够记住上一次调用的下标值。而且，每次调用函数时其下标值都重新设置为0。使用关键字static可以解决这两个问题，

```
void printAccounts()
{ static int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                          // increment index after use
```

设想一下，如果在每次调用时都将下标值重新设置为0，那么如何使下标值增大呢?

在printAccounts()的前一个版本中，每次调用时其初始值都赋给i。在这个版本中，由于i是静态的，它只赋值一次，尽管每一次调用都有赋值语句。实际上，赋值并不是在printAccounts()的第一次调用时进行的，它是在执行main()的第一条语句之前，在分配所有全局变量时进行的。当printAccounts()被调用时，会跳过其初始化赋值语句，因而使得这个局部变量的前一次的值可以用于下一次调用中。

```
void printAccounts()
{ static int i = 0;                  // executed only once
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                             // executed in each call
```

在这个例子中，显式地初始化操作也是不必要的，因为静态变量被隐式地初始化为0，因此printAccounts()的以下版本是完全合法的：

```
void printAccounts()
{ static int i;                 // implicit initialization to zero
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                        // executed at each function call
```

然而，维护人员应当花点时间去弄清楚为什么这个函数可以更新一个从来没有显式初始化的变量。虽然前一个版本不那么简洁，但它更好地传达了设计人员的意图。

使用局部静态变量并不是一个好的程序设计方法。它要求太多的客户和服务器函数之间的协调以及太多的精力去理解代码，而且它的用处不大。在大多数的情况下，找到一个不要求使用静态局部变量的解决方法并不难。例如，在程序6-3中（和前面一个版本的程序中）的账户打印方法是简单的，并且也不要求静态局部变量。

另外，静态全局函数是不能在其定义所在的文件之外调用的函数，因为在其他文件中一个静态函数的名字是不可见的——类似于静态全局变量。这意味着该函数名字可以在其他文件中使用，而不会出现名字冲突等问题。如果一个函数只能被位于同一个文件中的函数调用，那么一个好的方法是显式地将该函数定义为静态的，以使它只在所定义的文件中可见，而不是在整个程序中都可见。在程序6-3中，函数printAccounts()应该定义为静态的。

```
static void printAccounts()
{ static int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                             // increment index after use
```

类似地，在程序6-4中的函数printCaption[]也应该定义为静态的。

```
static void printCaption()          // called from this file only
{ cout << caption; }
```

类似于静态全局变量，这里的问题是名字冲突以及与维护人员之间的交流。通过把服务器函数与其调用者放在同一个文件中并把它们定义为静态全局函数，就可以允许开发其他程序文件的程序员使用这些函数名字而彼此之间不用进行协调。另外，它显式地向维护人员说明在其他文件中没有函数会调用这个函数。将服务器函数和客户函数放在同一个文件中不总是可能的，当这样做的时候，应该将服务器函数定义为静态的。

为了将函数绑定在类中，还有使用静态存储类别的另外两个方法。它们只能在类的静态存储区中存取数据。后面，将会讨论更多关于静态函数和静态域的知识。

6.3 内存管理：堆的使用

C++的作用域规则和各种存储类别为帮助程序员管理程序对象做出了很大的贡献。然而，这些机制不能充分地解决实现动态数据结构的问题。

用标志或者计数来实现动态数据结构的数组是简单的。当数组中的元素增加或减少时，用这些方法可以增加或者删除元素，然而它们需要在编译时知道数组的最大长度是多少。若确定的最大长度不适当，就可能会有溢出的危险或空间的浪费。

可以通过动态地分配和重新分配内存的动态内存管理解决这个问题。当需要把所有的变量都放在数组中时，就动态地分配一个更大的数组，把原来的数据拷贝到新数组中，并且释放原来的数组空间；当数组元素不断减少而浪费了太多的空间时，就分配一个小一点的数组，把原来的数组内容拷贝到新数组中，并释放原来那个大的数组空间。这个技术消除了溢出的危险和可能出现的空间浪费。

连续数组的另一个问题是，只有当新元素添加在数组的尾部时它们才是有效的。如果我们需要在数组的开头或中间添加一个新元素，就必须向数组的尾部移动其余的数组元素以腾出空间。

类似地，当从数组的中间而不是尾部删除一个元素时，我们必须向数组的头部移动其余的元素。这些操作要求额外的机器时间。另一种方法可以不必进行移动，而是引入一个标志去表示已删除的元素。这消除了在删除之后的移动，但在搜索元素时要求额外地测试每个元素的有效性。如果数组长度较短，或者在中间进行插入和删除操作不是那么频繁，那么上述的缺点都是无关紧要的。但对于较长的数组以及频繁的插入和删除操作，以上的处理可能会影响程序的性能。

对这些问题的另一种可能的解决方法是不立即为这些元素分配数组空间，而是在元素确实要插入到数据集中时，才为它分配内存。我们使用指针把元素链接起来，指针中含有这些动态分配的地址。通过指针，可以把一个元素插入数组中而不必花时间去移动其他元素。当该元素要删除时，它的内存就释放给其他变量使用，使用指针就可以不用移动其他元素或者标志元素已被删除。

在动态数组和链式数据结构中指针是很常用的。然而，它比我们前面讨论的长度固定的数组要更加复杂。处理指针时出现的错误是运行时错误而不是编译时错误，且通常难于发现。另外，频繁的内存分配和释放还可能会影响性能。

在很多语言中，比如Lisp、Eiffel和Java等语言中，认为内存管理对程序的完整性十分重要，因此这些语言不会将内存管理完全交给程序员去做，而是采用了所谓的自动内存回收技术，它估算程序对内存的使用并回收程序员应该归还的内存空间。当然，这些算法相对较慢、

较复杂和不精确。

在C++中，采用了相反的做法——但确是为了一个类似的原因！在C++中也认为内存管理对程序性能十分重要，因此它不信任一个普通的（或经常效率不高的）算法。在C++中，程序员具有对内存进行分配和释放的完全控制权。如果程序员犯了错误，就可能导致内存破坏或内存泄漏和程序崩溃，这是很糟糕的。但好的程序员不会犯太多的错误，动态内存管理的算法比较简单，只要我们细心地实现它们，它们是安全的。标准库也提供了适当的数据结构和函数帮助程序员避免发生错误。

用显式命令来分配动态数据的那块内存称为堆，该名字源于多次进行空间分配和释放的组织方法。使用堆结构便于搜索一块适当大小的内存空间来满足下一次的内存请求。

存放全局和静态变量的固定数据区域的大小是在编译和连接时计算出来的。而栈和堆的大小不能够准确地计算出来。通常，栈和堆都分配了较大的空间以避免过早的溢出。

分配堆中的变量，与一般的变量之间有两个不同点。

- 一般的变量（分配在栈和固定存储区中）是根据语言规则来分配的；而堆中的变量是由程序员用显式操作来分配的。
- 一般的变量有名字是为了作为内存单元的别名（用于方便记忆）；分配在堆中的变量没有名字，它们是通过指针来引用的。

6.3.1 作为类型变量的C++指针

指针是一个变量，它含有另一个变量（可以是一般的（有名字的）栈变量）的地址。然而，指针通常指向分配在堆中的变量（无名字变量）。指针自身通常分配在固定存储区中（作为全局的或静态的变量）或分配在栈中（作为自动的变量）。很少会将一个指针分配在堆中。指针是一般的名字变量。

在C++中，指针通常用于下列情况中：

- 运行时数组大小的动态分配。
- 创建由非连续的链接结点组成的动态数据结构。
- 向函数传递参数。

在本章中，我们将讨论指针的一般语法和语义，以及给出一些关于指针的前两种用途的例子。向函数传递参数的方法将在下一章中讨论。

为了创建一个指针变量，首先，必须指定它是一个指针；其次，必须指定该指针可以指向的变量的类型。一个C++指针只能指向（换句话说，是提供间接引用）一种类型的变量；该类型是在定义该指针变量时已确定的。当我们了解了C++更多的先进性能后，比如继承和多态（这里使用的两个术语还未进行解释），我们会看到这个规则的一些有意思的例外。但此时，要记住一个好的建议：定义为指向整数类型的指针应该指向一个整数而不是一个双精度数。类似地，定义为指向双精度类型的指针应该指向一个双精度数而不是一个整数。

为了表示某个变量是一个指针，我们可以在类型名之后或该变量名之前使用星号*；运算符周围的空格（或没有空格）是不重要的。

```
int * pi; char* pc; double*pd;          // any spacing is OK
```

注意，星号在这里不是运算符，它只是一个符号。它表示该变量是指向星号左边类型的一个指针。可从右向左读该指针声明。例如，pi是一个指向int类型变量的指针，或pc是一

个指向char变量的指针，等等。不久，我们会发现这样读也可以：*pi是int类型，或*pc是char类型，等等。

然而，在表达式中星号不仅仅是符号，它还是一个作用于一个指针变量（pi、pc等）去获取基本类型值的运算符。所获取的就是由该指针变量所指单元的值。获取由该指针所指值的星号运算符的名字是间接引用运算符（或称间接运算符）。

如果获得值的地址称为取址，则从地址获得值应称为析址（depointing），而不是间接引用。但C++从C借用了这个术语，原因是C是为汇编语言程序员而设计的一个高级语言，而汇编语言程序员习惯根据他们自己的喜好来称呼。

星号指针记号的作用域只是一个指针变量：它作用于跟在星号之后的标识符，而不是位于其前面的类型名。这不同于其他定义和声明的工作方式。例如，以下只有pc是指向char的指针，而pchar是char类型，而不是char*。

```
char* pc, pchar;          // pchar is of type char, not char*
```

这是一个相当容易混淆的问题，为了表示pchar也是一个指针，可以写为：

```
char* pc, *pchar;         // both pc and pchar are pointers
```

同样，指针变量是规则的有名字变量，存储类型是自动的或全局的，为了存放一个具体类型的地址而被分配在有效空间中。通常，指针的大小和整数的大小一样，但我们不应该对它进行计算，用运算符sizeof就可以得知所使用机器的情况，而依赖于指针大小来编写程序显然不是一个好方法，因为这样做的程序将是不可移植的。

警告　指针变量（地址）通常是整数大小的，不管它们指向的值是何种类型。不要在程序中使用指针的大小，因为它可能使程序不可移植。

类似于其他变量，指针在定义时不具有确定的值（如果是全局的变量则为0，如果是自动的变量则是不确定的）。一个指针只可以含有下列对象的地址：

- 基本类型（比如，char* pc）。
- 程序员定义的数据类型（比如，Account* pa），
- 基本的或程序员定义数据类型的数组（记号和变量一样，比如char* pc或Account* pa）。
- 函数（在这里暂时不讨论指针）
- 其他指针（比如，char** pcc。作为一个指向字符指针的指针；这是允许的，但我们不应该立即在程序中使用它）。

为了存取指针所指的对象的值，我们可以把间接引用运算符（星号*）作为一个一元运算符作用于该指针上（写在指针名的左边）。换句话说，可以间接引用该指针。例如，下面的语句把5.0移到一个由指针pd所指的double变量中，把20移到由指针pi所指的一个int变量中，如果pd所指的double值是正数（实际上它是正数）就把'a'移到pc所指的字符变量中。

```
*pd = 5.0; *pi = 20; if (*pd > 0) *pc = 'a';          // not ok
```

在前面曾经提过，如果pi是指向int类型的指针，那么*pi就是int类型；类似地，*pd是double类型。从值的类型的观点来说，这个例子是允许的。然而，由于从未对这些指针初始化，因此间接引用未初始化的指针是非法的。

当一个全局指针没有初始化时，它含有0。间接引用一个空指针会立即终止程序。

```
pd = NULL; *pd = 5.0;          // null pointer exception
```

当一个局部指针（自动变量）没有初始化时，它含有一个和其他所有自动变量一样的随机组合。这个组合可以解释为一个地址，但这个地址可以是内存中的任何地址。读取这个单元将返回垃圾；而对这个单元写内容可能会破坏计算机的内存。它可能使操作系统崩溃，引发运行时的内存保护异常，产生不正确的结果，尽管也可能产生正确的结果（如果没有程序用到指针所指的地址）。使用没有初始化的指针是一个常见的错误，且难以诊断，因为它们可以指向内存的任何区域。

未初始化的指针可以将我们带到内存中的任何单元，而这可能会导致内存破坏或出现不正确的结果。在C++中间接引用未初始化指针的这些错误是运行时错误，并非编译时错误这是不幸的：如果我们犯了错误，那个友好的编译程序不会告诉我们去改正错误，我们只能通过运行时测试来推测错误的存在。

通过使用取址运算符（引用运算符&），指针可以初始化为指向某个命名变量。程序6-5给出了一些关于指针操纵的例子。它的main()函数定义了两个类型分别为int和char的自动变量。它也定义了指向int和char的两个指针，将字符指针初始化为指向字符变量、将整数指针初始化为指向整数变量。在这之后，通过间接引用指针把一个新的值赋给整数变量，然后，通过间接引用指针来检查该整数变量的值，并通过间接引用字符指针赋字符值。最后，将字符指针指向该整数值。

程序6-5 对一般命名变量使用指针

```
#include <iostream>
using namespace std;

int main()
{
  int i; int *pi; char *pc;                    // noninitialized pointers
  pi = &i;                                     // this turns pointer to i
  *pi = 502;                                   // this is ok, but so is i = 502;
  if (*pi>0) *pc = 28791;                      // same as if(i>0) i=28791
  pc = (char*) &i;                             // some compilers don't need it
  int a1 = *pi;                                // access to i through pointer
  int a2 = *pc;                                // access to i through pointer
  cout << " i as decimal: " << i << endl
       << " i as hex: " << hex << i << endl;
  cout << " i through int pointer: " << dec << a1 << endl;
  cout << " i through char pointer: " << a2 << endl;
  cout << " i through char pointer in hex: " << hex << a2 << endl;
  return 0;
}
```

大多数编译程序不允许直接进行pc=&i的赋值；实际上pc的类型是char*，而&i的类型是int*。C++的以下要求也是严格的：它允许数值类型之间的隐式转换，但不允许不同类型的指针之间的转换。为了使指针赋值运算符有效，指针必须属于完全相同的类型。然而，不同类型的指针之间的显式转换（类型转换）是允许的，而且没有限制。现在，通过间接引用字符指针就可以存取和改变整数位模式中的位。图6-4给出了两个指向同一个整数变量的指针的间接引用。根据指针类型的不同，会得到不同的结果。

在程序6-5中，把hex和dec称为操纵器。它们向cout对象指明了计算输出值的基数（十

进制或十六进制）。类似于endl操纵器，它们被插到输出流中并改变它们的性质。从图6-4中可以看到，通过指针pi取回的值是正确的（28791），但由字符指针pc取回的值是不正确的。如图6-4中十六进制的输出所示，字符指针只取回了值i（十六进制数7077）的一部分（十六进制77）。请注意整数指针可以看到整个整数，但字符指针只能看到一个字节。例如，它们都不能正确地取回一个double值。因此保证间接引用正确类型的指针是很重要的。

```
i as decimal: 28791
i as hex: 7077
i through int pointer:   28791
i through char pointer:  119
i through char pointer in hex: 77
```

图6-4　程序6-5的输出结果（注意对int的不正确访问）

提示　当间接引用指针时，要保证指针类型和指针所指的类型是相对应的。否则，通过指针取回的值将是不正确的。

指针上的操作不太直观，很难通过阅读程序去理解指针运算。通过画图会有助于直观地了解指针的运算。我们可以画两种图：一种可以表明变量是分配在栈中还是在堆中（如图6-5a所示），另一种可以强调什么类型指针指向什么类型值（如图6-5b所示）。

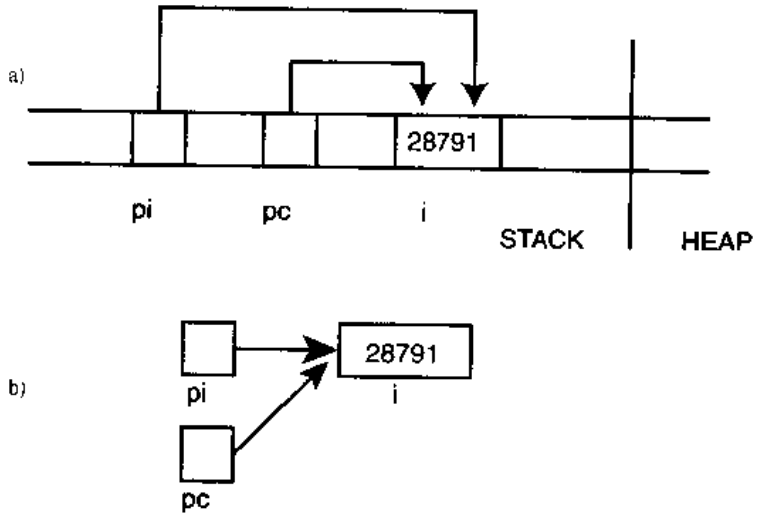


图6-5　整数指针和字符指针指向在堆栈上已分配的命名的整数变量i

图6-5a给出了分配在栈中的整数i、整数指针pi和字符指针pc。即使它们的实际大小可能是一样的，但常见的方法是将指针表示为更小的矩形。这里给出了整数i所含的值，而指针pi和pc含有i的地址，但我们不知道（并且不想知道）这个地址。因此不是使用地址，而是使用箭头来说明两个指针指向同一个单元。这里不关心指针是含有高字节的地址还是含有低字节的地址，只是想说明指针指向的值以及访问这些指针可以取回的值（假如指针的类型是正确的）。

图6-5b给出了同样的设置，它没有指定变量i、pi和pc是分配在栈中还是在堆中。如果变量的名字被指明，它们就分配在栈中；如果名字没有被指明就分配在堆中。同样，箭头表

示指针含有箭头所指的变量的地址；因此指针可以用来存取这些变量。

和我们见到的一样，对命名变量的操作使用指针并不是很有用。为了这种数据运算而使用指针并不比直接使用这些指针所指的变量（此例中是i）更好。把指针指向不合适的类型会导致复杂化和错误。一些程序员在函数调用中使用类似的技术去避免使用取址运算符（我们将在下一章中见到）。但这不是使用指针想要达到的目的，它们的使用是为了在堆中分配存储空间。

6.3.2 堆的内存分配

大多数C++的运算符是简单的符号。由于C++的运算符多于标准键盘上的特殊符号，C++提供了双符号运算符甚至三符号运算符。但这还不够，C++还使用了一些保留字作为运算符。其中的两个保留字是new和delete。它们都是一元运算符，只有一个操作数。这些运算符用于堆的内存管理中。在这里堆只是一个术语，程序员不一定要知道堆置于哪里。堆是什么？堆是通过使用运算符new和delete分配和释放内存的内存区域。我们所应该知道的就是，已经分配的内存应该在适当时候回收。

运算符new把类型名当做它的操作数[⊖]；它要求操作系统为指定的操作数分配其类型值所需要的内存空间。如果分配成功，运算符new就返回操作系统在堆中的内存单元地址。这个地址通常赋值给某个类型的指针，并且这个指针可以用来操作已分配的无名内存。如果系统内存已用完，运算符new就返回0而不是一个堆地址，程序还可以通过测试这个返回值来决定下一步干什么（例如，打印一条消息并终止）。

运算符delete把指针的名字当做它的操作数[⊖]。它在堆中找到指针所指的那块区域，然后要求操作系统将这块单元（其大小由在指针定义中的类型确定）标记为未使用。任何使用new运算符来分配内存的程序要含有一个回收内存的对称的运算符delete，以避免内存的泄漏，这是很重要的。

程序6-6给出了使用这些运算符的例子。它的main()函数定义了两个指针：pi指向整数类型，pc指向字符类型，然后用new来对它们进行初始化。在这之后，测试内存分配是否成功。如果不成功，程序就终止，因为它不能做它想做的事。通常，应该采取一些恢复措施去让程序温和地终止（保存数据）。有时，程序可能想释放一些内存并以有限功能的特殊形式继续执行下去。如图6-6所示，内存的分配是成功的，正确地设置了指针并返回了相应的值（整数28791和字符'a'）。

程序6-6 对无名堆变量使用指针

```
#include <iostream>
using namespace std;

int main()
{
 int *pi; char* pc;                          // noninitialized pointers
 pi = new int;                               // get unnamed space, point to it
 if (pi == NULL)                             // if new fails, it returns zero
  { cout << "Out of memory\n"; return 0; }   // or try to recover
 pc = new char;                              // get unnamed space, point to it
 if (pc == 0)                                // necessary precaution
  { cout << "Out of memory\n"; return 0; }   // or try to recover
 *pi = 28791;
```

⊖ 原文为“运算符”（operator）。——译者注

⊖ 原文为“运算符”——译者注

```
  if (*pi > 0) *pc = 'a';                          // manipulate unnamed objects
  cout << " integer on the heap: " << *pi << endl;
  cout << " character on the heap: " << *pc << endl;
  delete pi; delete pc;                            // part of heap memory life cycle
  cout << " (after delete) int: " <<*pi <<" char: " <<*pc <<endl;
  return 0;
}
```

```
integer on the heap:   28791
character on the heap: a
(after delete) int: -572662307 char:
```

图6-6　程序6-6的输出结果

在这个例子中，NULL是一个库常量。很多程序员更喜欢使用这个常量而不用数字0去表示该源程序正在处理的指针。如果使用数字0，其结果是一样的。重要的是要记住使用运算符new之后都要测试堆内存分配是否成功。

运算符delete把由运算符new分配的内存归还给堆。它有能力知道指针操作数的类型并释放与new运算符所分配空间一样大小的内存。如果忘记了使用delete，程序也能够工作，但随着时间的推移，特别是应用程序连续不断地工作时，程序可能会用尽堆内存，于是new将返回0。能够释放程序所申请的所有堆内存，是一种程序设计的技能。

当我们阅读含有delete运算符的程序时，可以读为“删除pi，删除pc”。但我们要知道这实际上并没有删除指针，删除的是指针pi和pc所指的（适当大小的）无名字的堆内存。这里的指针是有名字的栈变量，它们是根据本章前面所讨论的作用域规则来分配的。它们的内存是在指针定义执行时分配的（在这里是函数main()的开头），当执行超出了指针的作用域时，指针所占的内存就会释放，也就是当执行到达指针作用域的闭花括号处就会释放指针所占的内存（这里是main()的闭花括号）。

只能删除无名字的堆变量，而删除有名字的栈变量不是一个好的做法。例如，对于上面程序6-5中的变量i（通过指针pi或通过指针pc删除，或不通过指针直接删除）。

在删除了由指针所指的堆变量之后，该指针变量又变为未初始化的，因此不应该用来进行间接引用。在程序6-6的结尾，我们想取回由指针pi或pc所指的值，如图6-6所示，这些指针现在有可能指向任何单元，而不是它们原本应该指向的单元。注意，编译程序不会指出程序员所犯的指针操作错误。操作系统也不会阻止这类错误，虽然它们可以而且也应该阻止这类错误。

最后，关于delete运算符还有一点要注意：不应该对一个未初始化的指针使用这个运算符，它只能用在已由new运算符分配了堆内存的指针上。例如：释放同一内存两次是一个运行时错误（不是一个编译时错误）。

```
delete pi; delete pi;          // this code is incorrect
```

因为其行为未定义，所以该段代码是不正确的。它可能崩溃，也可能产生不正确的结果。要注意对内存管理的处理，特别是在循环中。不要在同一个指针上进行两次删除操作。对一个NULL指针进行删除是允许的，虽然它没有什么结果。

图6-7给出了程序6-6的内存图示。图6-7a说明了指针pc和pi分配在栈中，无名字的整数

值和字符值也分配在堆中。图6-7b也说明了相同的情形，指针是有名字的（分配在栈中），而整数和字符是无名的（分配在堆中）。我们试图概括地描述整数和字符的尺寸大小不同，但并不会对指针这么做。指针被画得比它们所指向的值要小，尽管有时它们占用更多的内存。

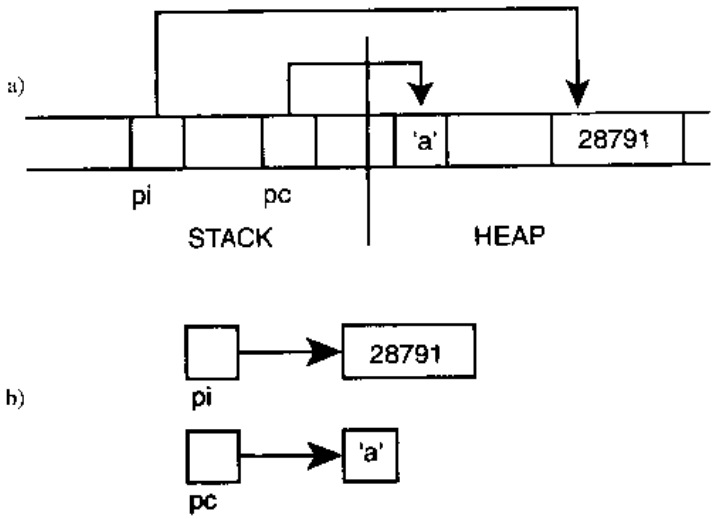


图6-7 整数指针和字符指针分别指向分配在堆中的无名整数变量和无名字符变量

运算符new和delete在C++中是可用的，但在C中却不能使用。在C中，动态内存分配是通过调用库函数malloc()来完成的，而内存的回收是通过调用库函数free()来实现的。函数malloc()的功能不如运算符new的功能，它不知道数据类型的大小，因此需要把所需的字节数作为参数。它也返回一个一般的、不能用于间接引用的所谓空指针。由malloc()返回的指针必须通过使用类型转换运算符转换为某个类型。如果内存分配失败，malloc()就返回NULL指针，于是程序可以检查所需要的内存是否可用。由于C++和C是兼容的，因此C++也支持malloc()，它定义在cstdlib（或stdlib.h）标准库中。

```
pi = (int*) malloc(sizeof(int));        // get unnamed heap space
```

函数free()把一个指针当做它的参数并释放相应的内存。

```
free(pi);                    // return heap memory for other uses
```

程序6-7给出了与程序6-5同样功能的操作实现，但它使用了malloc()和free()。注意头文件stdlib.h。这个程序的输出结果与图6-5相同。

程序6-7 使用malloc()和free()来进行内存管理

```
#include <iostream>
#include <cstdlib>                          // header for malloc() and free()
using namespace std;

int main()
{
 int *pi; char* pc;                         // noninitialized pointers
 pi = (int*) malloc(sizeof(int));           // get unnamed space
 if (pi == NULL)                            // if malloc() fails, it returns zero
   { cout << "Out of memory\n"; return 0; }  // or try to recover
 pc = (char*) malloc(sizeof(char));         // get unnamed space
 if (pc == NULL)                            // necessary precaution
```

```
  { cout << "Out of memory\n"; return 0; }   // or try to recover
 *pi = 28791;
 if (*pi > 0) *pc = 'a';                    // manipulate unnamed objects
 cout << " integer on the heap: " << *pi << endl;
 cout << " character on the heap: " << *pc << endl;
 free(pi); free(pc);
 cout << " (after delete) int: " <<*pi <<" char: " <<*pc <<endl;
 return 0;
}
```

实际上，很多C++编译程序是以函数malloc()和free()的形式来实现运算符new和delete的。然而在C++中，new和delete的使用比malloc()和free()更加频繁，因为它们更简单。当它们用来管理类对象的内存时，它们也可以隐式地调用特殊的函数、构造函数和析构函数。而函数malloc()和free()做不到这些。然而，这里有一个配对的问题。这些运算符和库函数必须成对地使用。如果内存是用new分配的，则不能用free()回收。同理，如果内存是用malloc()分配的，则不能用delete回收。注意，编译程序也不能发现这种错误，并且运行时测试也不起作用。为了避免出错，很多程序员只使用运算符new和delete，而不使用malloc()和free()。

然而，C（C++）程序中有一些使用malloc()和free()的重要函数。鉴于程序的寿命可能引起千年虫问题，应该准备好在未来的若干年中处理这些函数调用。

可见，对基本类型的值使用堆来进行动态内存管理是有趣的，但却没有多大的用处。我们可以通过使用栈中的有名字变量来代替动态内存管理。

有些程序员在堆中分配了单个的值。尽管这不是一个错误，程序能够正确地编译并执行，但是会把问题复杂化。单个的值的动态内存管理使得我们要考虑内存分配和释放的适当时间，并增加了程序中指针的定义、初始化和释放的复杂度。而所有这些工作都没有什么好处。应该避免这种做法，请记住，只有动态数组和动态数据结构才使用堆内存。

6.3.3 数组和指针

在编译时确定数组长度是C++的一个重要特点，这样做的目的在于充分利用内存和提高运行时的性能。我们知道，这样做会导致数组溢出或内存浪费的问题，因为很多应用的数组大小要到运行时才可以确定。而C++语法不允许使用不确定的值来定义数组的大小，因此才需要进行动态内存分配。

为了能够动态地分配数组，应该学习C++数组和指针之间的关系。这个关系建立在另一个C++的特性上：数组的名字（没有括号或其他修饰符）意味着第一个数组元素的地址。因此，可以用数组的名字来初始化某个类型（与数组元素类型相同）的指针，使得该指针内容变为数组第一个元素的地址，间接引用该指针将返回（或改变）数组的第一个元素。于是，可以将指针作为函数调用时的数组名字的同义词，并可以把它与数组下标一起使用。

在下一个例子中，定义了两个短的字符数组buf[]和data[]，以及定义两个指针p和q，并使用第一个数组元素的地址来初始化指针。这里，既可以通过使用地址符号来显式进行（p=&buf[0]），也可以通过使用数组名字（q=data）来隐式地进行指针的初始化。

```
char buf[6], data[6], *p, *q;       // arrays and pointers
int i;
p = &buf[0];                        // explicit syntax for address of first element
q = data;                           // implicit syntax for address of first element
```

```
for (i=0; i < 6; i++)              // assign array components
  { p[i] = 'A'+i;                  // uppercase letters "ABCDEF"
    q[i] = 'a'+i; }                // lowercase letters
```

指针和数组名字的不同在于，可以将指针重新赋值指向另一个单元（使用取址运算符&，或给指针赋值），但数组名字是一个常量因而不能重新赋值为另一个地址。在下一个例子中，数组data[]的小写字母部分被拷贝到数组buf[]的后半部分，使得数组buf[]含有字母“ABCabc”而不是“ABCDEF”。

```
p = &buf[3];                       // turn it to point to second half of the array
for (i=0; i < 3; i++)              // replace last 3 components
  p[i] = q[i];                     // same as buf[i+3]=data[i];
```

在这两个程序段中，指针名字都作为数组名字。所有使用q[i]的地方，也可以使用data[i]。这是一种好的想法，但不实用，因为它不允许我们做任何别的事情。当然，这只是我们对数组使用指针的部分工作。

C++另一个独特之处是，在指针上的算术操作与该指针所指的类型和内存元素的大小有关。例如，如果ptr是一个指向位于地址2000上的一个double类型变量的指针，那么ptr+1所指的地址是地址2008，而不是地址2001。

当指针指向数组元素时，就会特别方便。把指针加1并不像数值类型上的算术运算那样把1加到指针变量的值中，而是使指针指向数组元素的下一个元素！于是间接引用该指针就会返回（或改变）下一个数组元素的值！把指针加2就会把它移动两个元素的位置。在下一个例子中，数组data[]的前半部分（同样的小写字符“abc”）被拷贝到数组buf[]的前半部分，使得数组的内容变为“abcabc”。

```
p = buf;                           // point to start of the array again
for (i=0; i < 3; i++)              // replace the first half of array
  *(p+i) = *(q+i);                 // again, same as buf[i]=data[i];
```

注意，间接引用运算符的优先级高于算术运算符，因此不能写为*p+i，该式子表示了p[0]+i而不是p[i]。

对指针使用加1（或减1）运算符可以编写出更加简洁的程序代码。无论如何，指针加1实际上意味着把类型的大小加到存储在指针中的地址上，于是移动了指针，使之指向下一个数组元素。程序6-8概括了前面那些例子。在第一个循环中，它使用p[i]来代替buf[i]去设置和显示数组buf[]（ABCDEF）中的内容。在第二个循环中，它修改了数组的后半部分；在那个循环中，p[i]表示buf[i+3]而不是buf[i]。第三个循环使用一般符号来显示数组buf[]（ABCabc）。然后，指针p回到buf[]的开头。第四个循环代替了buf[]的前半部分。其结果是通过使用在指针上的加1运算符来显示的。程序的输出结果如图6-8所示。

程序6-8 使用指针来处理数组

```
#include <iostream>
using namespace std;

int main()
 {
  char buf[6], data[6], *p, *q;          // arrays and pointers
  int i;                                 // array index
  p = &buf[0];                           // explicit syntax for address
  q = data;                              // implicit syntax for address
  cout << "Initial buffer: ";
```

```
    for (i=0; i < 6; i++)                    // assign array components
      { p[i] = 'A'+i;                        // upper case letters
        cout << p[i];                        // display ABCDEF
        q[i] = 'a'+i; }                      // q and data are synonyms
    p = &buf[3];                             // point to second half
    for (i=0; i < 3; i++)                    // replace last 3 components
      p[i] = q[i];                           // same as buf[i+3]=data[i];
    cout << endl << "Replaced second half: ";
    for (i=0; i < 6; i++)
      cout << buf[i];                        // display ABCabc
    p = buf;                                 // point to start of array
    for (i=0; i < 3; i++)                    // replace the first half of array
      *(p+i) = *(q+i);                       // same as buf[i]=data[i];
    cout << endl << "Replaced first part: ";
    while (p - buf < 6)                      // incremented pointer
      cout << *p++;                          // dc not overuse this feature
    cout << endl;
    return 0;
}
```

```
Initial buffer:     ABCDEF
Replaced second half:  ABCabc
Replaced first part:   abcabc
```

图6-8　程序6-8的输出结果

当将加1和间接引用运算符用于同一个表达式时，比如*p++，它们的优先级是一样的，并且它们从右向左求值，而不是像大多数C++运算符那样从左向右地（见第3章的表3-1）进行运算。然而，后缀运算符的作用是在指针加1之前传送指针的值。因此，*p++的意思是：保存旧指针，指针加1并指向数组的下一个元素，然后返回由旧指针所指的地址上的值。换句话说，如果temp是一个字符指针，*p++等价于：

```
(temp = p, p++, *temp)
```

同样，指针和数组名字在许多方面都是等价的，但有一个例外：指针可以增大或重新赋值，但数组名字却不能。例如，在程序6-8的结尾处，这样打印数组buf[]将是一个错误：

```
while (p - buf < 6)          // displacement in array elements
  cout << *buf++;            // syntax error
```

为了实现这一功能，新增另一个指针就不会有问题：

```
q = buf;
while (p - q != 0)           // displacement in array elements
  cout << *q++;              // do not overuse this feature
```

注意指针p在这里作为一个岗哨值。在程序6-8的结尾处，指针p已增大为超过了数组buf[]的最后一个元素。因此当指针q指遍数组的所有元素并指向数组最后一个元素的下一个位置时，即和指针p指向同一个位置时，上面的循环就终止。

当然，理解指针符号和数组符号的关系是重要的。有很多遗留的C和C++程序都使用这个特点。指针符号不够直观，并容易使没有经验的人感到困惑。因此使用下标比使用指针更好。然而，利用指针而不是下标是编程能力成熟的一个标志，因为在指针上进行算术运算十分简洁。

以前，指针上的运算不仅简洁，而且可以产生更快的可执行程序。但由于有了现代编译程序，这不再是正确的了。无论用的是数组还是指针都能产生相同性能的代码。

6.3.4 动态数组

目前，已经讨论了指针的几种用途：作为有名字的栈变量的指针、作为无名字的堆变量的指针和作为有名字的栈数组的指针。这些技术没有给我们带来任何优点反而会使得我们的程序变得不必要的复杂。甚至，使用指向有名字的数组的指针（比如上述的例子）也不是很有用。使用有名字的数组比使用指针更简单。

现在讨论一些使用指针会显得有益和合适的例子。指针可以帮助我们动态地管理内存，并使我们在编译时不必确定数组的大小。通过使用动态分配的数组，我们可以达到此目的。

程序6-9给出了第5章的程序5-11中的程序的一个简化版本，它处理从键盘输入的事务数量。

程序6-9 可以防止数组溢出的事务数据输入程序

```
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
 const int NUM = 3;                          // for debugging: it should be larger
 double amount, total = 0, data[NUM];
 int count = 0;                              // initialize current data
do {                                         // do until EOF or array overflow
 cout << "Enter amount (or 0 to finish): ";
 cin >> amount;                              // get next double from the file
 if (count==NUM || amount==0) break;             // overflow or sentinel
 total += amount;                                // process current valid data
 data[count++] = amount;                         // and get next input line
 } while (true);
 if (amount != 0)                                // was all data read in?
  { cout << "Out of memory: input was terminated\n";
    cout << "The value " << amount << " is not saved" << endl; }
 cout << "\nTotal of " << count << " values is "
      << total << endl;
 if (count == 0) return 0;                       // no results if no input
 cout << "\nTran no. Amount\n\n";                // print the table header
 cout.setf(ios::fixed);                          // set up fixed format for double
 cout.precision(2);                              // total digits if NO ios::fixed
 for (int i = 0; i < count; i++)                 // go over the data again
  { cout << setw(4); cout << i+1;                // tran number
    cout << setw(11); cout << data[i] << endl; }  // tran value
 return 0;
}
```

在第5章的程序5-11中所用的技术（在数据输入的结束处使用一个字符岗哨值）是一种好的输入方法，于是在程序6-9中使用0标志。当输入数字0时，就终止循环的输入。如果用户输入数据的个数超过了数组data[]的大小，也终止该循环。

于是，循环输入可以在两种情况下终止：遇到输入的结束标志或者数组出现溢出。在这个例子中，程序将打印数组溢出的警告信息。检查是否count = = NUM并不可靠，因为输入数 据可能与数组中的元素正好一样多。尽管对于处理几百个或可能是几千个输入的实际程序来说，这不大可能会经常发生，但不进行边界检查仍可能会导致出错。

在程序6-9中，当循环终止时，就测试是否遇到了岗哨值（数字0）。如果不是，那么循环终止的原因就是数组溢出。如果遇到了岗哨值，那么我们就认为所有的数据都已读入了。

这个程序的输出结果如图6-9所示，其中数组只有3个输入。

```
Enter amount (or 0 to finish): 22
Enter amount (or 0 to finish): 33
Enter amount (or 0 to finish): 44
Enter amount (or 0 to finish): 55
Out of memory: intput was terminated
The value 55 is not saved

Total of 3 values is 99

Tran no.  Amount

   1       22.00
   2       33.00
   3       44.00
```

图6-9　程序6-9的输出结果（输入被删节）

这里的格式化数据输出结果与第5章的程序5-11不同。函数sett()设置cout对象的控制标志，它使用标志ios::fixed作为参数，表示该显示使用有小数点的定点数形式，而不是使用有尾数和指数的科学计数法。函数precision()以数字的个数作为它的参数。如果没有设置ios::fixed标志，这个数字表示所要显示的总的有效数字位数。当设置了ios::fixed标志后，这个数字表示了在小数点之后显示的数字位数。切记不要混淆函数precision()的这两种定义。

警告　函数precision()的参数意义依赖于ios::fixed标志的设置。设置了标志后，参数就表示在小数点之后要显示的数字位数。当没有设置标志时，它是总的有效数字位数。

在程序5-11的例子中，函数width()指定在显示中下一个输出值所占的最小位数。如果需要显示更多的位数，就会使用额外的位置（这就破坏了格式）。其他的格式化函数setf()和precision()也影响输出格式，只有用一个不同的参数重新调用这些函数，才会改变输出的格式。函数width()只有一个输出值作用域。在下一个值输出了之后，显示格式又变为缺省宽度（0），也就是说输出数据没有进行任何的格式化。因此在输出每个数值之前都要调用width()函数，才能进行格式化输出。

在程序6-9中，使用了操纵符setw()，将它插入到输出流中，像先前讨论的endl、dec和hex等操纵符一样。类似于函数width()，这个操纵符的作用域只是一个输出值。因此如果要保持域的宽度一致，就要将操纵符setw()插到每个输出值之前。注意，在程序6-9使用了头文件iomanip，前面的例子虽然使用了操纵符（至少endl），但它们不需要这个头文件，只有使用了参数的操纵符才需要这个头文件，比如setw()。如果忘记了把该头文件包含进来，程序将不能通过编译。

警告　当使用格式化函数（例如width()）或没有参数的操纵符（例如endl）时，不必把头文件iomanip包括进去。但使用有参数的操纵符（例如setw()）时，就应该把iomanip头文件包括进去。

程序6-10所实现的功能和程序6-9的一样，但前者是通过动态分配数组来实现的。这就是使用指针的好处，它把程序的集成性和执行的有效性结合在一起。

程序6-10 将数据读入到分配在堆的数组中

```
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
 const int NUM = 3;                                  // for debugging:it should be more
 double amount, total = 0, *data;
 int count = 0, size = NUM;                          // initialize current data
 data = new double[size];                            // initial array on the heap
do {                                                 // do until zero is entered
 cout << " Enter amount (or 0 to finish): ";
cin >> amount;                                       // get next double value
 if (amount == 0) break;                             // stop when sentinel appears
 if (count == size)                                  // out of space, ask for more
  { size = size * 2;                                 // make it conspicuous
    double *q = new double[size];                    // double array size
    if (q == 0)
     { cout <<" Out of heap memory: input was terminated" <<endl;
       break; }
    else {
    cout << "More memory allocated: size = " << size << endl;
    for (int i=0; i < count; i++)                    // copy old data
      q[i] = data[i];                                // use subscript notation
    delete [] data;                                  // do not forget to free old data
    data = q; }                                      // hook up main pointer
 total += amount;                                    // process current valid data
 data[count++] = amount;                             // and get next input value
  } while (true);
 if (amount != 0)                                    // and what is this for?
  { cout << "Out of memory: input was terminated\n";
    cout << "The value " << amount << " is not saved" << endl; }
 cout << "\n Total of " << count << " values is "
      << total << endl;
 if (count == 0) return 0;                           // no results if no input
 cout << "\n Tran no. Amount\n\n";                    // print the table header
 cout.setf(ios::fixed);                              // set up fixed format for double
 cout.precision(2);                                  // total digits if NO ios::fixed
 for (int i = 0; i < count; i++)                     // go over the data again
  { cout << setw(4); cout << i+1;                    // tran number
    cout << setw(11); cout << data[i] << endl; }     // tran value
 return 0;
}
```

这里不是使用预定义的编译时常量（在本例中是3）在栈中分配数组data[]，而是通过确定数组大小的变量size，在堆中分配相同大小的数组。这个变量具有一个运行时的值而不是一个编译时的值。注意动态数组没有名字，它只是通过字符指针data来存取。当指针用来指向有名字数组（见程序6-8）时，可以使用指针名字（比如p[i]）或数组名字（比如buf[i]）。在这里由于堆数组没有名字，因此只能使用指针去存取数组的元素。

如果程序输入的下一个数可以放入数组中，也就是条件count==size仍为假时，该数值就保存在数组中。注意指针在这里作为数组名来使用。程序6-10和程序6-9中的以下语句data[count++]=amount; 是相同的，但其含义不同，在程序6-9中，data是一个栈数组

的名字；在程序6-10中，data则是指向一个无名字的堆数组的指针名字。

当动态数组已填满并使条件count==size变为真时，事情就会变得有趣。在程序6-9中，会发出一个出错的消息，然后停止数据的进一步输入。而在程序6-10中，通过在堆中分配更多的内存并将原来的数值复制到一个新的堆数组中，就可以使数组从溢出中恢复过来。

程序不能使用同一个指针data去分配更多的内存。当data指向较大内存块的地址时，它就失去了现有数据的内存地址。因此要使用另一个局部指针q以便在堆中分配另一个数组，这个数组是已有的堆数组大小的两倍。使内存大小加倍是一个常见的堆管理策略，但也可以使用其他增大内存的方法。

```
double q = new double[size*=2];       // get more heap memory
```

在分配内存的语句中修改size不是一个好的方法。维护人员可能很容易略过这个操作。更好的做法是在使用运算符new之前显式地进行分配。

```
size = size * 2;
double *q = new double[size];       // double array size
if (q == NULL)
  { cout << "Out of heap memory \n"; return; }
else
  /* copying data into array pointed to by q */
```

注意 一个指针类型变量double*可用来指向double类型的一个单值和double类型的一个数组。一般来说，只有指针类型是不能够确定它是指向一个数组还是一个单值的。这使得向维护人员传达设计人员的意图变得更加困难。

一个好的方法就是每次都检查内存分配是否成功。内存用完的情况是很罕见的，因此程序员通常都不检查运算符new是否返回0。当将已存在数组复制到新数组的前半部分时，指针的名字可以作为数组名字，这样就避免了指针上的运算。

```
for (int i=0; i < count; i++)
  q[i] = data[i];          // copy old data into the first half of new data
```

一些程序员会将下标与指针算术运算一起使用，比如用下面的方式：

```
for (int i=0; i < count; i++)       // copy old data
  *(q+i) = *(data+i);
```

还可以使用下面的循环，它使指针加1并指向堆数组中的下一个位置：

```
for (double *p=q,*r=data; p-q < count; p++, r++)     // is it nice?
  *p = *r;
```

还可以使用下面的形式：

```
double *p = q, *r = data; int i = 0;
while (i++ < count)
  *p++ = *r++;                          // real nice?
```

本书建议尽可能地使用数组符号并避免指针算术运算，但是其他形式的循环也是合法的C++用法。你会在遗留的C/C++代码中见到它们。

回到关于动态内存分配的问题。在将原来的数据复制到新数组之后，原来的数组必须删除，并且指向原来数组的指针应该转到指向新数组。以上的步骤和它所产生的结果都是十分重要的。如果不删除原来的数组，程序就可能会出现内存泄漏。如果先把指针转到指向新数组，将不能够删除原来的数组。

程序6-10的运行结果如图6-10所示。

```
Enter amount (or 0 to finish): 22
Enter amount (or 0 to finish): 33
Enter amount (or 0 to finish): 44
Enter amount (or 0 to finish): 55
More memory allocated: size = 6
Enter amount (or 0 to finish): 66
Enter amount (or 0 to finish): 0

Total of 5 values is 220

Tran No. Amount

  1       22.00
  2       33.00
  3       44.00
  4       55.00
  5       66.00
```

图6-10 程序6-10的输出结果（带有调试消息）

在程序6-10中的程序中有很多小问题。注意它只有一个出口可以离开循环：当运算符>>不能读入下一个输入时。那么，在循环结束处的if语句有什么用呢？这是程序设计中的一个常见的不明智之举：把不再有用的语句留在源代码中。这样做的后果是，维护人员测试毫无意义的语句所花费的时间，常常超过理解有意义的语句的时间。

程序设计随意性的另一个例子是变量amount的定义。为了避免名字冲突，特别是在维护时，以及为了方便维护人员理解代码，尽可能在嵌套块结构中的内层定义变量，是很重要的。因此指针q定义在if语句的局部块中。我们不能对变量total、data、count这样定义，因为它们需要用在输入循环之外。但变量amount可以定义在输入循环之内，这看起来不是一个主要的问题，因为该程序是那么小；但在一个适当的地方进行定义是一个重要的编程技能，我们应该通过实践把它掌握好。

当该程序终止时，我们没有把多余的内存还回给堆。在这个例子中这可能并不危险，因为操作系统将会进行清理；但这不是一个好的程序设计风格，而且依靠于操作系统也是不好的设计。

不归还内存就有可能危及内存溢出。应该养成这样一个习惯：在堆中分配了内存以后，应在一个适当的地方将内存返回给堆。在以下的情况中，main()函数应该在return语句之前立即执行。

```
delete [] data;          // array (but not the pointer) is deleted
```

注意在删除数组时的delete语句中的中括号。由于运算符delete不清楚data是指向一个数组还是单个值，因此，这里需要用中括号去表示删除的是数组而不是单个变量。在权衡程序员与编译程序之间的利益时，C++再一次偏向于方便编译程序。

当在回收原来的数组之前把它复制到新数组时，要使用count而不是使用size作为循环的限制，尽管这段代码以测试count==size为开头。在复制时并不存在这个相等关系，因为在分配新数组之前已经增大了size。因此，在复制结束后再改变size可能会更好：

```
if (count == size)                              // out of space, ask for more
 { double *q = new double[2*size];              // double array size
   cout << "More memory allocated: size = " << size << endl;
   for (int i=0; i < size; i++)                 // copy old data
```

```
    q[i] = data[i];
  size *= 2;                                    // double the limit for next test
  delete [] data;                               // do not forget to free old data
  data = q; }                                   // hook up main pointer
```

图6-11概括了从数组溢出中恢复的所有操作。图6-11a给出了出现溢出时的堆数组data[]和栈变量amount、size以及count的情况（data[]就是指由指针数据所指的堆数组）。图6-11b给出了从数组data[]复制了值之后的堆数组q[]以及将空间还回给堆之后的数组data[]。图6-11c给出了同时指向新的堆数组的两个指针data和q。图6-11d给出了当指针q根据作用域规则被删除了之后的数组。

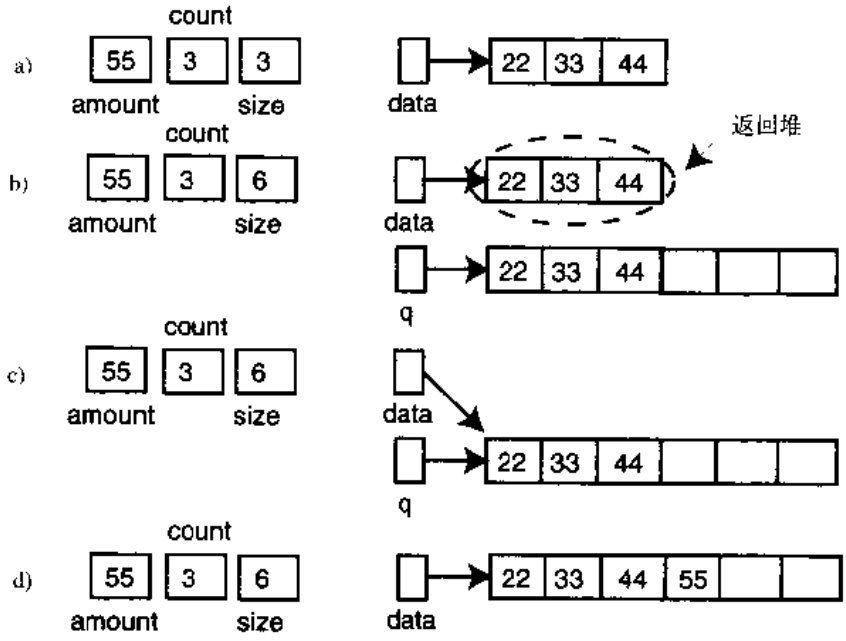


图6-11　从堆数组溢出中恢复的一连串动作

另一个有用的动态数组的例子与文本输入有关。当程序期待字符数据的输入时，例如输入顾客名字或地址之类的信息时，很难想象该程序需要一个多于30~50个字符的数组去容纳其输入。但如果键盘上的一个键被粘住了（由于咖啡外溅，或者任何其他的理由），会出现什么情况呢？输入数据将使这个小数组溢出并破坏内存。而且当我们从一个文件或一条电信线路中读入信息时，也无法限制输入数据的长度。内存破坏的危险总是存在的，不管程序分配了多大空间的数组。

程序6-11给出了这个问题的一个解决方法。其思想是把数据输入到栈中一个较短的有名字数组（buf[]）中，然后将数据复制到一个堆数组中（由指针data所指）。如果不断有数据输入进来，就继续把它读入栈数组中（覆盖已复制到堆数组中的数据），然后分配一个更大的堆数组（由指针temp所指），将以前的堆数组（由指针data所指）中的数据复制到这个更大的数组中，并接着添加栈数组buf[]中的数据。

程序6-11　使用一个动态数组去容纳无限的输入串

```
#include <iostream>

using namespace std;

int main(void)
```

```
{
 const int LEN = 8; int len=1;                    // short array for debugging
 char buf[LEN], *data = 0;                        // init to zero for first pass
 cout << " Type text, press Enter: \n";
 do {
   cin.get(buf,LEN);                              // data goes into a stack array
   len += strlen(buf);                            // total length of old data
   char *temp = new char[len];                    // request new heap array
   if (temp == 0)                                 // test for allocation success
     { cout << " Out of memory: program terminated\n";
       return 0; }                                // no luck, give up
   if (data == 0)
     strcpy(temp,buf); }                          // copy data from input buffer
   else
   { strcpy (temp, data); strcat (temp, buf); }   //  copy data
   delete [] data;                                 // delete existing array
   data = temp;                                    // point to the new array
   cout << " Total: " << len << " added: " << buf << endl;
   cout << " Dynamic buffer: " << data << endl;    // debug

 char ch = cin.peek();                             // what is left in the buffer?
 if (ch == '\n')                                   // quit if it is new line
  { ch = cin.get(); break; }
 } while (true);                                   // or keep going until EOF
 cout << "\n You entered the following line: \n\n";
 cout << data << endl;                             // same syntax as for arrays
 delete [] data;
 return 0;
}
```

在程序6-11中，用户把数据输入到一个栈数组buf[]中，数组的大小设置得较短（LEN=8个字符）以便说明算法是如何工作的。函数get()不断读入数据，直到它读入了LEN-1个字符或在输入流中发现了换行字符为止，它没有从输入流中删去换行字符（换行字符需通过其他方法删除，例如：调用只读入一个字符的函数get()）。

在任何情况下，get()都把一个终止符0加到buf[]的字符串中。下一步，程序将在堆中分配len个字符以便容纳读入buf[]的字符。第一次，设置len=len+strlen(buf)，因为len被初始化为1，而strlen()计算了除终止符0之外的字符个数。如果内存分配不成功（指针temp设置为0），程序就会终止。

如果这是第一次通过循环（指针data仍初始化为空），那么程序就到此结束：程序从buf[]中将输入数据复制到由temp所指的数组中。在这里，利用了指针和数组名之间的等价性，通过把指针temp传递给函数strcpy()，就可以从buf[]中将字符复制到由temp所指的数组中。

如果这不是第一次通过循环（指针data不是0；它指向前面分配在堆中的数组），程序的执行就相对复杂了。首先，程序使用strcpy()把前面的数据复制到新分配的数组中，然后使用strcat()把buf[]中的字符添加到前面数据的后面。

现在，指针temp指向更新后的输入串，而指针data指向前面的数据。下一步，程序删除前面的数据并使用指针data指向更新后的输入串。

下一个任务是弄清楚在调用get()时发生了什么：输入的终止是因为发现了换行字符（并留在输入流中），还是因为输入了LEN-1个字符而使得数组buf[]变满了？为了确定是

哪种情况，程序通过调用函数peek()来查看下一个输入字符；如果下一个字符的确是换行字符，程序就通过调用只读入一个字符的函数get()删去它，并终止循环。

如果输入行在缓冲区中放不下，位于do循环开始的函数get()就在读入LEN-1个字符之后终止，并且仍然把空终止符添加到数组buf[]的尾部。函数peek()将返回不是换行字符的下一个字符。因此，从输入流中删去这个字符不是一个好办法，因为它将是下一次调用get()时读入的第一个字符。

在下一次的do循环中，在循环体前面的get()语句把下一组字符输入到数组buf[]中；在这次循环中，我们必须从buf[]中把数据复制到一个动态分配的数组中。

这一次，由字符指针data所指的数组已经存在。它的长度（包括0终止符）存放在变量len中。程序使用局部指针temp在堆中分配另一个数组，它要求有足够的内存去容纳已存在的堆数组（由指针data所指）和在buf[]中新输入的字符。因此要计算表达式len+=strlen(buf)。程序为新分配的数据组赋值：先把data所指的数组复制进来，再将它和数组buf[]的内容串接起来。此后，程序就删除由data所指的原来的数组，并将data指向新分配的数组（由temp所指向的）。循环继续进行，直到在peek()的下一次调用中发现了换行字符，或者发现了文件的结束符为止。

程序的执行结果如图6-12所示。

```
Type text, press Enter:
Hello World!
Total: 8 added: Hello W
Dynamic buffer: Hello W
Total: 13 added: orld!
Dynamic buffer: Hello World!

You entered the following line:

Hello World!
```

图6-12　程序6-11的输出结果（带有调试消息）

同样，这里分配在堆中的数组没有名字。它们是通过分配在栈中的指针来引用的。（因此，这些指针有名字temp和data）。程序把这些指针看做是分配在栈中的数组来使用。例如，以与一般数组buf[]完全一样的方式将指针temp和data传递给函数strcpy()、strcat()和strlen()。对于程序6-11结尾处的插入运算符<<也有同样的道理：将指针data看做是数组名字一样地使用。这里的区别是，有名字数组的内存是在它们的作用域结束处根据语言规则而回收的，而动态变量所占的内存是通过使用显式的delete运算符（注意在delete语句中的中括号）回收的。

图6-13给出了对图6-12中的输入数据的内存管理操作。图6-13a说明了满载了"Hello W"时，指针data是0（用接地符号表示）。图6-13b说明了变量len的值为8，temp指向一个有8个字符的堆数组，data也指向同一个数组。（注意，在一个空指针上进行的delete运算符没有什么影响。）图6-13c说明了在输入"orld!"之后的数组buf[]。图6-13d说明了len的值为13，temp指向含有"Hello World!"的数组，然后删除掉由data所指的数组，最后data指向与temp一样的数组。

注意　在第一次删除指针之后，没有对指针初始化就对它第二次使用delete运算符是非法的。然而，将delete运算符作用于一个值为0的指针是完全合法的。这样的

操作没有产生不良的影响。

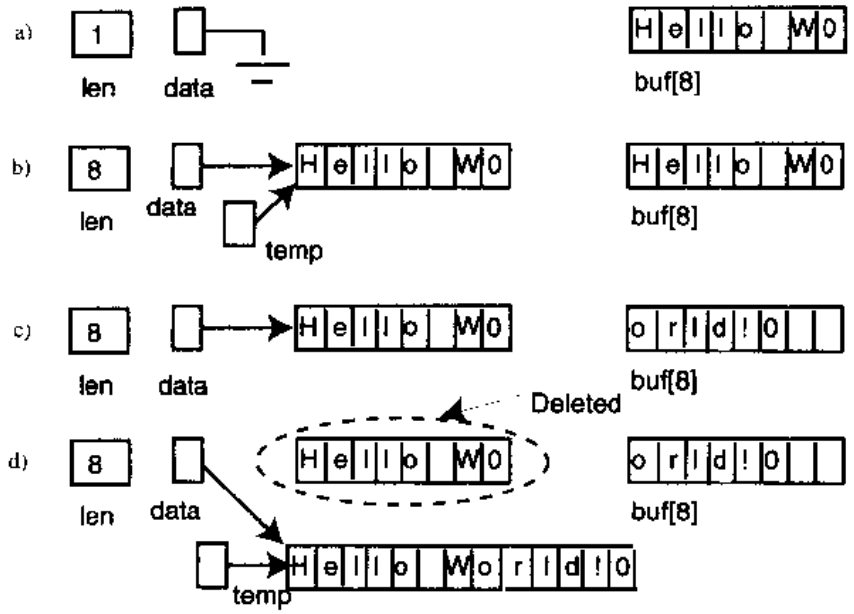


图6-13 图6-12中输入数据的指针示意图

不知道大家第一次阅读这个问题时会花上多少时间。如果觉得事情变得越来越复杂了，我们可以跳过这段内容。等我们有了编程以及调试等经验之后，内存管理会变得相对简单一些。

如果对这段内容感觉可以接受，就可以继续读下去。前面例子处理的是输入一行任意长度的输入数据。对于像C++之类的要求固定大小的栈数组的语言来说，可以将以上的技术用于很多实际应用中。

下一个例子以前面的讨论为基础：它输入任意数目的、任意长度的行。我们将在本章的下一节中看到写磁盘文件的技术。

程序6-12将程序6-11作为一个内层循环，外层循环继续读输入数据，直到用户按了Enter键并且不再继续输入任何字符为止。这个空行可作为终止输入的标志。

程序6-12 使用一个动态数组去输入任意多行

```
#include <iostream>

using namespace std;

int main(void)
 {
   const int LEN = 8; char buf[LEN];
   int cnt = 0;
   cout << "Enter data (or press Return to end): \n";
do {                                    // start of outer loop for input lines
   char *data = new char[1]; data[0] = 0; // initially, it is empty
   int len = 0;                         // initial size is zero
   do {                                 // start of inner loop for line segments
     cin.get(buf,LEN);                  // get next line segment
     len += strlen(buf);                // update total string length
   char *temp = new char[len+1];
   strcpy(temp,data); strcat(temp,buf);
   delete data;
```

```
    data = temp;                              // expand the long line
    cout <<"Allocated " << len+1 <<": " <<data <<endl;
    char ch = cin.peek();                     // what is left in the buffer?
     if (ch == '\n' || ch == EOF)             // quit if if new line
      { ch = cin.get();                       // but first remove it from input
        break; }
  } while (true);
    if (len == 0) break;                      // end on empty string
    cout << " line " << ++cnt << ": " << data << endl;
    delete [ ] data;
     } while (true);                          // continue until break on empty line
     return 0;
  }
```

程序6-11和程序6-12之间有几个有趣的差别。在程序6-11中，变量len表示分配在堆中的数组的大小。在程序6-12中，变量len表示被拷贝到堆数组中的字符个数；数组的大小还要加上1以便容纳0终止符。

这两个程序在处理第一次输入时也有所不同。第一次输入buf[]与其他次输入buf[]之间有两个差别。第一次输入时，堆数组尚未存在。因此内存的大小要求比输入buf[]中的字符个数多1，而不是等于堆数组与buf[]中的字符总数。于是要使用if语句。

```
if (data == 0)
  len = strlen(buf) + 1;      // first time copy from but[] only
else                          // otherwise copy from data[] and buf[]
  len = strlen(data)+strlen(buf)+1;
```

在第一次输入时，堆数组只是从buf[]中接收数据；在其他次的循环中，新分配的堆数组从已存在的堆数组以及数组buf[]中拷贝数据。因此程序6-11含有以下的if语句。

```
if (data == 0)
  strcpy(temp,buf);           // first time copy from buf[] only
else                          // otherwise copy from data[] and buf[]
  { strcpy(temp,data); strcat(temp,buf); }
```

然而，程序6-11不含有第一个if语句。程序员通常会觉得额外的测试会使程序难于理解，因此他们利用为不同情况而工作的数据来避免这些额外的测试。在程序6-11中，将data初始化为0，将len初始化为1。因此，可以把两种情况归结为以下的语句：

```
len = len + strlen(buf);      // works for first and for next read
```

这条语句需要仔细的解释和测试。上述的if语句是自解释性的。我们更喜欢什么？是长的自解释性代码还是需要解释的简洁代码？在前面几章我们曾说过其一，这里再说其二。

在程序6-12中采用了另一个方法，指针data开始时指向一个大小为1的堆数组，它的第一个（惟一的）字符是0终止符——因此它是一个空字符串。变量len被初始化为0：这是空字符串的长度。

```
int len = 0;                                  // initial length of data
char *data = new char[1]; data[0] = '\0';     // empty string
```

现在，第一次循环和所有其他次循环之间没有差别了：可以把buf[]的长度加到data[]的长度中，把data[]拷贝到新的堆数组中（第一次它是一条空字符串），然后添加上buf[]的内容。

```
do {                                          // start of inner loop for line segments
 cin.get(buf,LEN);                            // get next line segment
```

```
len += strlen(buf);                          // update total string length
char *temp = new char[len+1];                // allocate new heap array
strcpy(temp,data); strcat(temp,buf);         // merge data there
```

但请注意：有着两条if语句的版本会更容易自解释。

另一个问题是关于程序的终止。如果用户按下回车键，在程序6-12中的程序就应该终止。用户很可能会输入换行符'\n'（ASCII码10），而由peek()的调用返回它，于是终止内层循环。len--0的测试会终止外层循环。但在有些计算机上结果并不是这样。当输入字符然后按回车键时，就输入了换行字符。但当按回车键而没有输入任何字符时，输入的就是“文件结束”的标志。因此在程序6-12中，既要测试换行字符又要测试EOF（该常量的值是-1）。

```
if (ch == '\n' || ch == EOF)          // quit if it is new line or EOF
 { ch = cin.get(); break; }           // but first remove it from input
```

顺便提一下，程序6-11中的程序没有这样做。这意味着，如果我们按下Enter而不是输入一个长的字符串，程序就会陷入死循环中，因为没有换行字符使if语句为真。这么好的程序中存在如此严重的错误，真是太可惜了。

程序6-12也并不是很完美。当然它比程序6-11的要好一些，因为看起来它不会陷入死循环。但它将变量ch的类型定义为char，然后它把这个变量和EOF比较，而EOF是负数，这只有在char被看做是有符号数时才会起作用。但是字符类型是有符号的吗？对于不同的机器，情况不同。为了观察会发生什么情况，我们可以用下面的语句来代替ch的定义：

```
unsigned char ch = cin.peek();        // on end of file, it is 255
```

运行程序6-12中的程序，我们将看到它的确陷入一个死循环。

这是一个常见的可移植性问题。一个好的解决方法是使用整数类型。

```
int ch = cin.peek();                  // on end of file, it is -1
```

对于程序6-12中的程序，作者已作了完善。该程序的运行如图6-14所示。

```
Enter data (or press Return to end):
First line
Allocated 8: First l
Allocated 11: First line
 line 1: First line
This is the last line
Allocated 8: This is
Allocated 15: This is the la
Allocated 22: This is the last line
 line 2: This is the last line

Allocated 1:
```

图6-14 程序6-12的输出结果（带有调试消息）

现在是讨论内存管理的其他技术的时候了。如果大家觉得前面的内容已经很复杂，可以直接转到第7章继续学习。但以后不要忘了回过头来学习有关动态结构的内容。

6.3.5 动态结构

在前面一节中，我们讨论了运行时在栈内存中分配数组的方法，它不必在编译时确定数组的大小。其中使用指针的技术比在本章开始时讨论的那些技术要有用得多。

动态数组的使用消除了内存破坏和空间浪费的问题。但如果程序员没有正确地管理堆内

存，则会出现某些额外的复杂度和内存泄漏的危险。另外，还要考虑对程序性能的影响，因为使用堆内存会浪费机器时间。对于大多数的应用来说，动态数组管理对性能的影响并不明显，但仍然可能由于太频繁地分配和释放内存而减慢程序的速度。

所有的数组，无论是大小固定的还是动态数组，只有对数组的最后一个有效元素进行操作时，元素的添加和删除才会最快速和简单。当元素处于数组的中间而要进行插入或删除操作时，事情就会变得复杂。对于需要在中间频繁地插入和删除元素的数组来说，动态分配的结构是一个好的选择。

程序员定义的结构可以作为单个的结点分配内存，并连接到链表或结点网。为了能够组成一个链表，结点必须至少由两个成分组成：一个是信息项，另一个是下一个结点的地址（一个指向在链表中下一个结点的指针）。信息项可以是一个单值或者是一个结构，结构可以具有应用所需要的多个域。为了将注意力集中在程序设计问题上，我们将考虑一个非常简单的结构：信息项只含有一个值，比如事务数目。

在定义结点类型时，可以对含有下一个结点地址的域任意地命名。不妨称之为next。但这个域的类型就不能随便定义了。

```
struct Node {
  double amount;              // information item
  Node* next; } ;             // link to next Node
```

不论我们对结点用什么类型名字，next域的类型名字都一样，另外还要使用指针符号，因为next域是一个指向Node类型结构的指针。也可以对不同的情况使用相同的结点类型。为此，必须引入另一个类型Item，并使用typedef来定义它。（另一种方式是使用C++模板；它们更加灵活和复杂。）

```
typedef double Item;          // Item is synonym for double
struct Node {
  Item item;                  // information item
  Node* next; } ;             // link to next node
```

我们可以在任何时刻在堆中分配结点，并且只有当程序要存储一个新项目的信息时才分配结点（在从键盘、文件或网上输入数据之后）。因此，没有必要预先保留内存；也就是说，没有必要使用数组。这样就消除了为了不浪费空间而可能产生的数组溢出的危险，也不必为了插入或删除操作而移动元素。

链式结构的动态内存管理比使用动态数组更加复杂，它是一种重要的程序设计技术。用来进行结点运算的指针是有名字的变量；它们可以作为全局变量或者某个作用域（函数或块作用域）的局部变量而分配在栈中。对指针应该能够：a)适当地定义，b)适当地初始化，c)适当地管理。

程序员很少会对地址的值感兴趣。使用指针的目的不是想搞清楚地址在哪里，而是想通过使用指针的名字，而不是对象的名字（因为对象被分配在堆中而没有名字）去存取指针所指的对象。

这段代码说明了一个典型的程序设计的错误：它正确地定义两个指针，但却间接引用了还未初始化的指针。

```
Node *p,*q;                   // the scope of the * is one name
q->item = amount;             // it damages location pointed to by q
```

一个未初始化的指针可以指向任何地方。如果程序没有使用这块内存区域，其结果可能

是正确的。如果这块地方为操作系统的另一个程序所使用，就会引起无法预料的麻烦。

至此为止，我们讨论的只是右值需要初始化。在以下这段代码中，变量x必须在赋值运算（作为右值）之前被初始化，但变量y在使用之前不必初始化，因为变量y是左值。

```
int x; int y;              // definitions of noninitialized variables
y = x;                     // x needs initialization, but y does not
```

在上面的例子中q->item用作左值，因而不必初始化；然而，q用为右值，因此用它存取item域之前必须有一个合法的值。

当一个指针，比如q，被适当地初始化时，其内容是一个Node类型结构的内存地址。我们不知道指针q是否含有item域或next域的地址（或者含有结构的开始地址，结尾地址或中间地址）。因而找出它并用其结果来优化程序并不是一个好的做法。指针的地址应该保持为地址的抽象。不管q的内容是什么，*q都是由这个指针所指的值，在此例中它是一个Node类型的无名字的结构。存取该指针也称为间接引用该指针。同样地，q->item是指针q所指的结构中item域的值，而q->next是同一结构中next域的值。通过指向结构的指针（q->item和q->next）来存取一个无名字结构的域，也称为间接引用该指针。

一些程序员不喜欢使用选择运算符：箭头和圆点。而使用(*q).item代替q->item以及用(*q).next代替q->next是允许的。圆括号在这里是必需的，因为选择运算符的优先级高于间接引用运算符，因此，*q.item表示了*(q.item)，而这将是一个语法错误，因为圆点选择运算符只能用在结构变量上（有名的或无名的），而不能用在指针上。

程序不应该引用一个尚未初始化的指针。如果这个指针是全局的，则它的缺省值是NULL；而间接引用一个NULL指针将会是一个运行时错误，通常它会终止程序的运行。如果这个指针是局部的，则其值是垃圾。由于被解释为一个地址，这个值可以指向内存中的任何地方（堆或非堆）。确保能避免内存的破坏或不正确值的查找。

有几种方法可以设置指针的值。一种方法是通过使用取址运算符把一个有名字变量的地址赋值给指针，比如q=&count，这种方法不是很有用。我们还有其他两种方式设置指针的值：

- 在堆中分配一个新的无名字变量，并使用运算符new使得指针指向该变量（而我们不知道它是指向分配的内存的开头还是结尾）。
- 用一个已指向某个内存区域的指针作为赋值运算中的源；这个指针可能是：a)一个栈变量，b)一个堆变量的一个域。

这就是所有的指针初始化和赋值运算。下面是使用了两种指针初始化方法的代码段：

```
Node *p,*q = new Node;     // q is initialized, but p is not
q->item = amount;          // it saves value of amount in heap memory
q->next = NULL;            // popular sentinel for linked lists
p = q;                     // p now points to the same node as q
```

在很多算法中，都需要遍历一个链结构，也就是访问每一个结点并执行某些操作（取出一个项的值，检查是否达到最后一个结点，等等）。其中一种做法是使用类似于遍历数组的结点计数方法。另一种做法是不断遍历这些结点，直到发现岗哨值为止。一种标准的方法是将数组中最后一个结点的next域设置为NULL。这种方法的优点是这个NULL值不可能与一个指针的其他取值相混淆。就像我们先前提到的那样，用一个规则的0也能完成；但很多程序员更喜欢利用一个库定义值NULL来提示程序涉及指针的处理。

C++不允许将一种类型的变量地址赋值给指向另一种类型变量的指针。在此，C++是一种

强类型语言。在以下这段代码中，程序员想把Node变量（由指针q所指）的每一个字节的内容都以ASCII码字符的形式输出。注意，Node域中的所有编码并不都是可打印的字符编码。

```
char *c = q;                                    // no, this is a syntax error
for (int i = 0; i < sizeof(Node); i++)          // go over each byte
  cout << *c++ << ' ';                          // print each byte as a character
```

如果我们想打印结构的每一个字节，只需告诉编译程序（和维护人员）我们正在使用不同的指针类型。这里要使用C++的类型转换，如下所示：

```
char *c = (char*) q;                            // now this is NOT a syntax error
for (int i = 0; i < sizeof(Node); i++)          // go over each byte
  cout << (int)(*c++) << ' ';                   // print each byte as an integer
```

注意char类型和Node类型是不相容的。即使使用类型转换，一种类型的值也不能转化为另一种类型的值。这是C++支持强类型的特性。不同类型的指针之间不能够直接互相赋值，但使用了显式类型转换后，它们可以互相转换。

当程序员创建一个链式结构（在循环中）时，将会在堆中分配结点空间，然后将数据（从键盘或文件）填入结点的信息项中，并将结点链入链式结构中。链式结构有多种形式。这里考虑的是一个简单的链表，其中每一个新结点将添加到链表的尾部。

在链表结构中，程序可以依次地存取每一个结点，通常从链表的第一个结点开始，接着是下一个结点，直到遇到next域中含有岗哨值的那个结点为止。然而，问题是当插入新结点时，怎样将结点链入到链表的尾部。从开头开始遍历每一个结点以便找到含有标志的结点的过程很复杂，而且也没有必要。如果链表变得很长的话，搜索的时间对于插入操作来说可能太昂贵了。

对这个问题的一个解决方法是提供一个指向链表最后一个结点的指针。当创建了一个新结点时，就可以直接将新结点添加到表尾而不必逐个访问表中的结点。“添加”是什么意思呢？它表示最后一个结点（含有NULL地址的那个）的next域将设置为指向新结点。因此，我们需要表示最后一个结点的next域的名字（赋值运算的左值）以及新结点的地址（赋值运算的右值）。但两个结点都分配在堆中，因而它们没有名字！因此我们必须找出指向这两个结点（最后一个结点和新结点）的指针。在下面的代码段中，指向最后一个结点的指针是last，而指向新结点的指针是q。因此，把新结点添加到链表尾部的赋值运算是last->next=q;在上下文中，就应该是：

```
Node *last;                     // pointer to the last node
do {                            // do until EOF causes failure
   . . .                        // read value of amount from file
  Node* q = new Node;           // create new node on the heap
  if (q == 0)                   // test for success of request
    { cout << "Out of memory: input terminated" << endl;
      break; }                  // gracefully terminate if not
  q->item = amount;             // fill node with program data
  q->next = NULL;               // sentinel value for list end
  last->next = q;               // attach as last node in list
   . . . .                      // whatever else is needed to be done
} while (true);
```

这是一个好的解决方法。它给出了快速地把新结点插入到链表中而不必遍历表中所有已有结点的方法。但是这里没有说明如何实现两个重要的操作：怎样开始和怎样结束。怎

样开始的意思是怎样将第一个结点插入到一个空表中。怎样结束的意思是怎样保证下一次循环中指针确实指向表中的最后一个结点而不是以前的那个最后结点（现在位于新插入的结点的前面）。

当第一个结点插入到空表中时，表达式last→next没有什么意义，因为表中还没有结点，因而next域不指向任何已分配的结点。这意味着当第一个结点插入到空表中时，不应该进行以上的赋值运算，而应该在表头添加第一个结点。

通常表头由另一个指针指向，不妨称之为data。表示表中没有结点的一种方法是对表中的结点计数。当数目为0时，新结点应该由表指针data指向。当数目不为0时，新结点应该插入到表的尾部，即由last→next指向。

```
Node *last, *data; int count=0;                          // last/first pointer, node count
do {                                                     // do until until end of data
   . . .                                                 // read the value of amount
  Node* q = new Node;                                    // create new node on the heap
  if (q == 0)                                            // test for success of request
    { cout << "Out of memory: input terminated" << endl;
      break; }                                           // gracefully terminate if not
  q->item = amount;                                      // fill node with program data
  q->next = NULL;                                        // sentinel value for list end
if (count == 0)          // for the first node only
   data = q;             // attach as the first node in list
else
  last->next = q;        // attach as last node in list
 . . . .                 // whatever else is needed to be done
} while (true);
```

还记得条件运算符吗？这里可以利用条件运算符。该表达式是返回data还是last->next，依赖于cout的值，从而也决定是将data还是last->next赋值为q。

```
(count == 0 ? data : last->next) = q;          // nice code
```

另一种处理链表开始的方法是将链表指针data初始化为NULL。在循环中，分配和初始化了一个新结点之后，就测试链表指针是否为NULL。如果是，新结点就应该作为第一个结点并让data指向。如果表指针不是NULL，就意味着新结点不是第一个结点，并且应该插入到last->next。

```
if (data == NULL)          // this means that there are no nodes yet
   data = q;               // point the list pointer to the first node
else
   last->next = q;         // attach new node to the last list node
```

如果喜欢条件运算符，也可以写为：

```
(data == 0 ? data : last->next) = q;           // concise code
```

图6-15举例说明了这个问题。图6-15a给出了表的初始状态，指针data被初始化为0而指针last（现在）可以指向任何地方。图6-15b给出了插入第一个结点（amount的值是22）之后的链表状态：新结点已经初始化并由指针q指向，将指针data和last设置为指向新结点。注意将next域画得与指针data和last一样大的原因，是因为它们都有着相同的类型Node*。图6-15c给出了链表和另一个为了插入而分配的结点：它由指针q指向。图6-15d给出了在表尾插入的第一步：最后一个结点的next域（last->next）设置为指向新结点（由指针q所指）。

图6-15d给出了插入操作的第二步：指针last转为指向新结点。

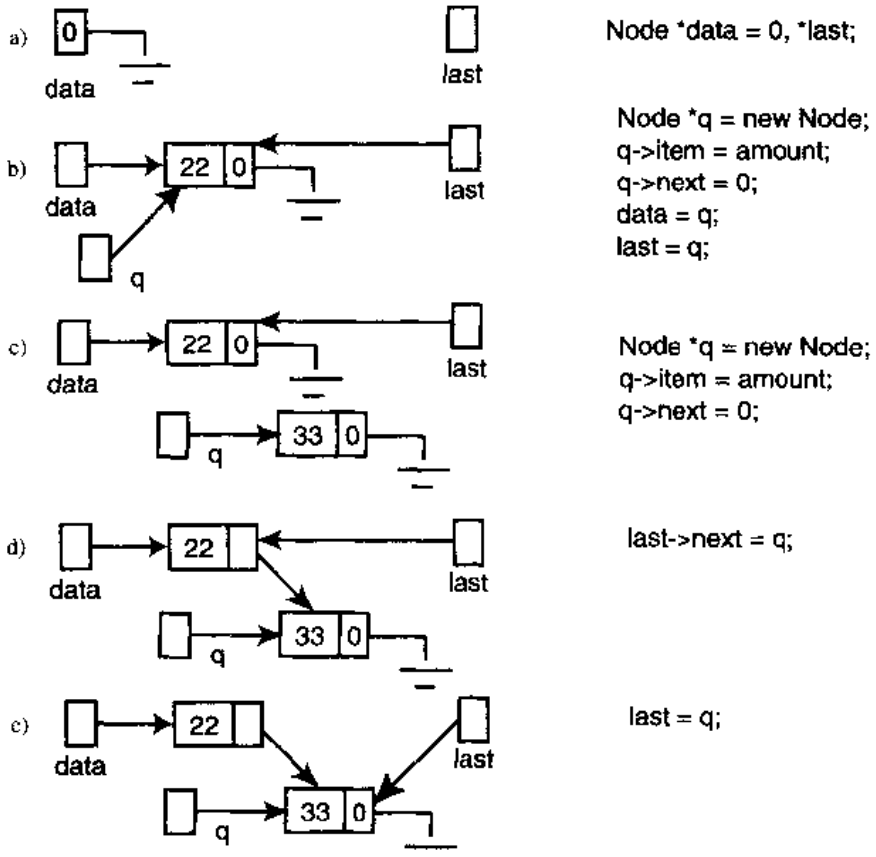


图6-15　在链表的尾部插入一个新结点的指针示意图

当新结点插入到表尾之后，就应该移动指针last，因为它指向最后一个结点前面的那个结点，因此last->next的赋值运算在下一次循环中将是不正确的。移动指针last意味着要设计一条赋值语句，其中指针last位于左边。那么赋值运算的右边应该是什么呢？为了回答这个问题，应该找到一个指针，它指向所希望的赋值目标应指向的结点，也就是一个指向新结点的指针。

在图6-15d中，是否存在指向新插入结点的指针呢？当然有。实际上，有两个指针指向那个新结点。一个是用来分配新结点的指针q。另一个是将该结点插入到表中的last->next指针，它们两个都可以使用。

```
last = q;          // divert the pointer back to the last node
```

使用指向新结点的第二个指针，可以有下列的语句：

```
last = last->next;        // move pointer to next list node
```

移动指针last的第二种格式实际上就是在链表中移动一个遍历指针，以便指向下一个结点的一种常见技术。这种技术在链表处理算法中非常流行。它等价于在数组中增加下标以便找到数组的下一个元素的语句i++。

程序6-13类似于程序6-9和程序6-10。事务数据从键盘读入，但不是在栈中分配一个固定数组（如程序6-9）或在堆中分配一个动态数组（如程序6-10），而是为读入的每一个值分配一个单独的结点。然后，将该结点插入到链表的尾部。

程序6-13 使用堆结点的链表

```
#include <iostream>
#include <iomanip>
using namespace std;
typedef double Item;
struct Node {
 Item item;
 Node* next; } ;

int main ()
{ int count = 0;                                  // count of amounts
 Node *data=0, *last;                             // pointers to start and end of list

do {                                              // do until EOF causes failure
 double amount;                                   // local variable for input
 cout << " Enter amount (or 0 to finish): ";
 if (amount == 0) break;
 cin >> amount;                                   // get next double from user
 if (amount==0) break;                            // stop input on no more data
 Node* q = new Node;                              // create new node on the heap
 if (q == 0)                                      // test for success of request
  { cout << "Out of heap memory" << endl;  break; }
 q->item = amount;                                // fill node with program data
 q->next = NULL;                                  // sentinel value for list end
 (data == 0 ? data : last->next) = q;
 last = q;                                        // last=last->next; is ok, too
 count++;                                         // increment count
  } while (true);
 cout << "\nTotal of " << count << " values are loaded\n";
 if (count == 0) return 0;                        // no output if no file input
 cout << "\nNumber Amount Subtotal\n\n";         // print header
 cout.setf(ios::fixed);                           // fixed format for double
 cout.precision(2);                               // digits after decimal point
 double total = 0;                                // total for input amounts
 Node *q = data;                                  // start at start of the list
 for (int i = 0; i < count; i++)          // go over list data
  { total += q->item;                     // accumulate total
    cout.width(3); cout << i+1;           // transaction number
    cout.width(10); cout << q->item;      // transaction value
    cout.width(11); cout << total << endl;      // running total
    q = q->next; }                        // idiom to pointing pointer to next node
 Node *p = data, *r = data;               // initialize traversing pointers
 while (p != NULL)                        // go on until it runs off the list
  { p = p->next;                          // prevent next node from hanging
    delete r; r = p; }                    // delete node, catch up with next
return 0;
}
```

在输入了所有数据之后，程序遍历链表。对于每一个结点，它打印每个事务的金额和事务总数。程序的输出如图6-16所示。

用来扫描链表的局部指针q被初始化为指向表的开头（q=data;），然后使count逐步递增，每一步，存取q所指的结点（在这个例子中，它累加total、打印事务的Number、Amount和Subtotal），然后通过将q设置为q->next遍历下一个结点。当q变为NULL时，表示已指向了最后一个结点（其next域是NULL），于是循环终止。

```
Enter amount (or 0 to finish): 22
Enter amount (or 0 to finish): 33
Enter amount (or 0 to finish): 44
Enter amount (or 0 to finish): 55
Enter amount (or 0 to finish): 66
Enter amount (or 0 to finish): 0

Total of 5 values are loaded

Number   Amount   Subtotal

  1       22.00      22.00
  2       33.00      55.00
  3       44.00      99.00
  4       55.00     154.00
  5       66.00     220.00
```

图6-16　程序6-13中代码的输出结果

这个循环的另一种方式是在for循环的开头设置遍历指针。

```
double total = 0;                              // total for input amounts
int i = 0;                                     // start at start of the list
for (Node* q=data; q!=NULL; q=q->next)         // go over list data
  { total += q->item;                          // accumulate total
    cout.width(3); cout << i+1;                // transaction number
    cout.width(10); cout << q->item;           // transaction value
    cout.width(11); cout << total << endl;     // running total
    i++; }                                     // increment the count of nodes processed
```

注意名字q已用于程序代码中，它是输入循环中的局部变量，因此它可以在后面的程序中继续使用。如果将它定义在main()函数的作用域中，类似于data，那么在程序中进一步使用它时，就要分析怎样使用这个指针：它是否能够用于其他的处理，以及是否应该引入一个不同的名字来代替它。要注意正确使用作用域的概念。

程序6-13的最后一个循环给出了遍历链表的另一种方式。其目的是将表结点空间退回到堆中，以避免内存泄漏。对于这个简单的例子来说，程序只是分配了结点，对它们遍历了一次，然后终止，这对内存的影响不大。该操作系统将会关注堆内存。对于在执行期间要进行多次（随着时间的变化）分配和释放结点的程序来说，对内存的影响会很大。对于这些程序来说，不能够释放那些不再需要的结点将会出现麻烦。

这里给出了另一种遍历链表的方法，循环应该扫描每一个结点并删除它。开始，应该初始化一个指针，使之指向第一个表结点，然后将指针移到另一个结点。当指针指向最后一个结点时，向下一个结点的移动使得这个指针成为NULL。使用for循环来实现链表的循环遍历是广泛采用的方法。

```
for (Node *q = data; q != NULL; q = q->next)  // visit each node
  { delete q; }                               // release its heap memory
```

这是一个好循环；它的开头很标准，可以用在很多情况中。但这个循环有一个问题：计数表达式q=q->next是在执行循环体之后测试循环终止条件之前执行的。然而循环体释放了由指针q所指的内存。这个内存可以用于其他目的而不会再被这个程序使用。这个循环在删除q之后要做的事居然就是让q指向q->next！

顺便提一下，在作者的计算机上，程序被正确地执行。因为指针q所指的内存只被标志为可利用的，而实际上还没有用于其他操作，因此表达式q->next实际上正确地取出了下一个

结点的地址。但这个程序是不正确的，在其他计算机上，这个程序会出错。

程序6-13使用了一种更加复杂但更加强壮的循环格式。开始时，令指针p和r指向同一个结点，运行时使p指向下一个结点，删除r所指的结点，然后使p和r再指向同一个结点。

```
Node *p = data, *r = data;      // initialize traversing pointers
while (p != NULL)
 { p = p->next;                 // move it to point to the rest of list
   delete r;                    // delete node, make pointer invalid
   r = p; }                     // catch up with the rest of list
```

注意将“删除r”读为“删除由r所指向的结点”而不是“删除指针r”。恐怕有些程序员可能认为这条语句会删除指针，但事实胜于雄辩。这个指针有一个名字，因此分配在栈中，根据语言规则，它是在这个变量定义所在的作用域的闭花括号处被删除的。运算符delete只删除堆中分配的无名字的变量。

最后，值得注意的是，当程序处理分配在堆中的链结点时，即使对所有的测试数据、程序都能正确地编译和执行，这也不意味着程序就是正确的。如果算法没有使用堆内存，就没有这种危险。

6.4 磁盘文件的输入和输出

在前面所有的例子中，程序从键盘读取输入数据并将数据输出到屏幕上。这是一种好的处理方法，因为它使我们可以在某个时间集中精神干一件事。对于实际应用来说，应该能够读取由其他应用所产生的数据，并且为了进一步的使用而保存其结果。在这一节中，我们将简单讨论处理大量数据的另一种技术，即文件。

类似于其他现代语言，C++没有固有的输入和输出操作。用来进行I/O操作的函数属于某个库而不是语言本身。C++程序可以使用两个库，从C继承的标准I/O库stdio和专为C++设计的新iostream库。

两个库都支持文本的输入和输出。C的库复杂且易出错，掌握它对于维护遗留下来的C程序的程序员来说是重要的。C++的库不那么容易出错，但仍然复杂和麻烦。为了理解C++库是怎样工作的，必须知道怎样使用C++的类、继承、多重继承和其他还未讨论的一些概念，因此这一节只讨论一些最简单的方法，它们能够使我们对磁盘文件读写数据。

6.4.1 输出到文件

让我们从写一个文件开始，因为这比读文件要简单一点。

实际上，把数据写到一个磁盘文件类似于把数据写到显示器屏幕上，但不是使用预定义的对象cout，而是使用类库ofstream（输出文件流）的一个程序员定义的对象。使用这个类定义的源文件必须包含头文件fstream。

在第2章已经提过，一个对象是把数据和行为结合在一起的类的一个实例，也就是说，它是一个其成员含有函数的结构。类库ofstream的设计思想是，所有预定义对象cout可以利用的函数，都可以被程序员定义的ofstream类的对象所利用。

于是，我们所要做的就是在程序中定义ofstream的一个对象，用它代替cout对象，以便将程序输出送到一个磁盘文件而不是屏幕上。其输出语句（包括格式化语句）与对象cout的输出语句一样完成同样的工作：将程序变量的位模式转换为要输出的一连串字符，但它是

输出到文件中而不是屏幕上。

在程序6-14中，重新实现了程序6-12。该程序输入一行任意长度的数据并保存到data.out文件中，所做的改动很小。

程序6-14　使用一个动态数组输入任意行数据，并把数据输出到一个磁盘文件中

```
#include <iostream>
#include <fstream>                          // for ifstream, ofstream objects
using namespace std;

int main(void)
 {
  const int LEN = 8; char buf[LEN];         // short buffer for input
  int cnt = 0;                              // line count
  ofstream f("data.out");                   // new: output file object
  cout << "Enter data (or press Return to end):\n";
do {                                        // start of outer loop for input lines
  int len = 0;                              // initial length of data
  char *data = new char[1]; data[0] = '\0';
  do {                                      // start of inner loop for line segments
     cin.get(buf,LEN);                      // get next line segment
  len += strlen(buf);                       // update total string length
  char *temp = new char[len+1];
  strcpy(temp,data); strcat(temp,buf);
  delete [] data; data = temp;              // expand the long line
  int ch = cin.peek();                      // what is left in the buffer?
  if (ch == '\n' || ch == EOF)              // quit if it is new line or EOF
     { ch = cin.get(); break; }             // but first remove it from input
 } while (true);                            // continue until break on new line
 if (len == 0) break;                       // quit if the input line is empty
 cout << " line " << ++cnt << ": " << data << endl;
 f << data << endl;                         // save data to the file
 delete [] data;                            // avoid memory leak
 } while (true);                            // continue until break on empty line
 cout << " Data is saved in file data.out" << endl;
 return 0;
}
```

可见，这里定义了一个命名为f的类ofstream的对象。当创建ofstream文件对象时，指定用来作为输出文件的物理磁盘文件名字为参数。

```
ofstream f("data.out");          // open output file
```

这条语句把对象f和放在可执行程序文件同一个目录下的物理文件data.out联系起来。如果需要一个在不同目录下的文件，就应该使用相应的路径名（记得使用'\\'去表示文件路径中的转义字符）。如果这个名字的磁盘文件不存在，就创建它。如果这个名字的文件已存在，就先删除原来的文件，然后创建具有相同名字的新的空文件。（支持文件版本的操作系统就创建该文件的下一个版本。）

如果磁盘已满或有写保护，情况会怎样呢？那么创建操作就只能是失败，它没有产生任何运行时错误。

处理这个问题的一种方法是调用成员函数fail()，如果前一次I/O操作失败（不管什么原因），它会返回真，否则返回假。

```
ofstream f("data.out");          // open output file dat.out
```

```
if (f.fail())                    // test for success, give up if not
   { cout << "Cannot open file" << endl; return 0; }
```

很多程序员会认为处理一个满或写保护的磁盘是很罕见的，因而忽略对这个可能性的测试。注意，忽略这个问题的程序是不可移植的。

在ofstream文件对象成功创建之后，它可以处理存入物理文件中的变量，其处理方式与cout对象处理要显示的变量的方式是一样的。这意味着当调用插入运算符<<时，计算机内存的数值位模式就被转换为一连串字符。对于字符数据，这个转换是不重要的。

```
f << data << endl;          // write array to output file, not to cout
```

可见，存取数据的语法和对象cout的方法是一样的。输出操作会失败吗？很多程序员觉得如果文件成功打开，就没有必要去检查每一个单独输出操作。这是不对的。记住，我们是在讨论存储大量的数据，即使现代磁盘与过去的相比有着巨大的容量，它们也有可能变满实际上溢出一个软盘甚至一个压缩盘是一点也不罕见的。因此需要在每次I/O操作之后测试它是否成功。

```
f << data << endl;                    // save data to the file
if (f.fail())                         // test for success of operation
  { cout << "Disk is full, output terminated" << endl; break; }
```

注意 假定I/O操作不会失败是草率的。文件对象的创建和信息写入之后总是应该接着进行测试，测试该操作实际上是否成功。

图6-17给出了程序6-14的运行例子。对于图6-17中的输入数据，其输出文件data.out含有下列行。

```
Enter data (or press Return to end):
First line
 line 1: First line
Second line
 line 2: Second line
This is the last line of text
 line 3: This is the last line of text

 Data is saved in file data.out
```

图6-17 程序6-14代码的执行例子

```
First line
Second line
This is the last line of text
```

当ofstream文件对象超出了作用域（在程序6-14中是在main()函数的结尾）时，就删除它。于是就切断了文件对象和物理文件之间的联系，然后物理文件关闭。注意，ofstream文件对象的删除并不会引起物理文件的删除。

6.4.2 从文件输入

现在来讨论其他例子，其中的程序使用由另一个程序、文本编辑程序或通信线路提供的数据。一个简单的做法是定义类ifstream（输入文件流）的一个对象来表示输入文件流。

类似于类ofstream，类ifstream也定义在包含头文件fstream的源文件中。同样类似于类ofstream，物理磁盘文件的名字也用来作为对象的参数。

```
ifstream f("amounts.dat");      // open file amounts.dat for input
```

如果指定的文件找不到会怎样呢？或者，另一个应用程序正在使用它而不能被其他程序打开，会怎样呢？类似于ofstream，该ifstream对象仍然被创建，但它不能用来进行输入。任何创建一个ifstream对象的尝试之后都应该对成功与否进行测试。

```
ifstream f("amounts.dat");          // open file amounts.dat for input
if (f.fail())                       // test for success
  { cout << "Cannot open file" << endl; return 0; }
```

当ifstream类型的文件对象定义成功时，对象的名字就和物理磁盘文件的名字建立了关联。在这之后，就可以使用抽取运算符>>将数据输入到程序变量中。这时，不是使用表示键盘的对象cin，而是使用程序员定义的文件对象f。存取数据的语法和cin对象的语法一样，所有其他的输入函数get()、getline()、setf()以及precision()和操纵符都可以使用，并且使用的方式完全相同。

注意，当使用抽取运算符时，输入一系列的字符并将它们转换为指定类型的位模式（如果转换可行）：整数、双精度、字符等。抽取运算符会跳过前面的空格（包括换行字符）直到遇到所转换的字符为止，当遇到不是该值一部分的内容（比如，换行字符）时就停止。它也可以以二进制形式而不是一连串字符形式从文件中读入数据。二进制形式会更加紧凑，但文本编辑程序无法读入它或以一个可读的形式显示在屏幕上。

一个输入操作会失败吗？当然可能。而且，当我们从一个输入文件读取数据时，希望该操作在程序到达文件尾部时最终停止。为了检查是否已到达文件的结尾，可以使用成员函数eof()，如果到达文件结尾它会返回true，否则它会返回false。

```
do {                                // do until EOF causes failure
  double amount;                    // local variable for input
  f >> amount;                      // get next double from file
  if (f.eof()) break;               // stop input on no more data
```

注意，前面的语句有点模糊。“到达文件的结尾”是什么含义呢？这里有两种可能的解释，并且我们应该知道它们的差别。当程序从一个文件读取数据时，在程序读取文件中的最后一项之后就立即出现文件结束的条件。另一种可能是，只有当程序想越过文件的最后一项进行读取时才出现文件结束的条件。

Ada和Pascal采用第一种解释。在这些语言中，一个从外部文件读取数据的do循环应该形如（用C++语法来写）：

```
do {                                // Ada or Pascal loop structure
  if (f.eof()) break;               // stop input on no more data
  double amount;                    // local variable for input
  f >> amount;                      // get next double from file
  . . . . . }                       // process the amount read
```

COBOL、C++和Java采用第二种解释：只有当程序想越过文件中的最后一个输入进行读取时才出现文件结束的条件。在这些语言中，一个从外部文件读取数据的do循环结构应该是不同的。

```
do {                                // C++ or Java loop structure
```

```
double amount;                    // local variable for input
f >> amount;                      // get next double from file
if (f.eof()) break;               // stop input on no more data
. . . . . }                       // the rest of the loop
```

如果我们在一个C++程序中使用了第一种循环结构，会发生什么情况呢？最后一个值将从文件中读取并由循环的余下部分处理。在下一次循环中，eof()将返回false，于是语句f>>amount;将又执行一次。当没有数据时，将出现文件结束的条件，但内存中amount的值仍然是一样的（在大多数系统中）。由于程序不知道已经没有数据，循环的余下部分将继续处理最后一个值，好像它在输入文件中出现了两次一样。在下一次循环中，将出现文件结束的条件，于是循环终止。

警告 在C++中，文件的结束符在程序从文件中读取最后一项时并没有出现。它是在下一次读取时，即当程序想越过最后一个文件项读取时才出现。要避免两次使用最后一个文件项。

程序6-15给出了程序6-13的这个版本，它从文件中而不是从键盘读取数据。为了便于比较，将从键盘读取的那些语句改为了注解，而没有删除它们。可见，从键盘读取转到从文件读取是不难的。图6-18给出了程序执行的结果。

程序6-15 为了从磁盘文件中读取数据而使用堆结点的链表

```
#include <iostream>
#include <iomanip>
#include <fstream>                                  // for ifstream class
using namespace std;

typedef double Item;

struct node {
Item item;
Node* next; } ;

int main ()
{
  int count = 0;                                    // count of amounts
  Node *data=0, *last;                              // pointers to start and end
  ifstream f("amounts.dat");                        // file to read data from
  if (f.fail())
  { cout << "Cannot open file" << endl; return 0; }
do {                                                // do until EOF causes failure
  double amount;                                    // local variable for input
// cout << " Enter amount (or 0 to finish): ";
// cin >> amount;                                   // get next double from user
// if (amount == 0) break;
  f >> amount;                                      // get next double from file
  if (f.eof()) break;                               // stop input if no more data
   Node* q = new Node;                              // create new node on the heap
   if (q == 0)                                      // test for success of request
    { cout << "Out of heap memory" << endl; break; }
   q->item = amount; q->next = NULL;                // fill node with data
   (data == 0 ? data : last->next) = q;
   last = q;                                        // last=last->next; is ok, too
   count++;                                         // increment count
   } while (true);
 cout << "\nTotal of " << count << " values are loaded\n";
```

```
  if (count == 0) return 0;                          // no output if no file input
  cout << "\nNumber Amount Subtotal\n\n";            // print table header
  cout.setf(ios::fixed);                             // fixed format for double
  cout.precision(2);                                 // digits after decimal point
  double total = 0;                                  // total for input amounts
  int i = 0;
  for (Node *q = data; q != NULL; q = q->next)       // OK
   { total += q->item;                               // accumulate total
     cout << setw(3) << ++i;                         // transaction number
     cout << setw(10) << q->item;                    // transaction value
     cout << setw(11) << total << endl;              // running total
   }
  Node *p = data, *r = data;
  while (p != 0)
  { p = p->next;                                     // return heap memory
    delete r; r = p; }
  return 0;
}
```

```
Total of 4 values are loaded

Number  Amount  Subtotal

  1     330.16     330.16
  2      76.33     406.49
  3      50.00     456.49
  4     120.00     576.49
```

图6-18　程序6-15中代码的执行例子

用来产生图6-18的文件amount.dat含有下列行

```
330.16
76.33
50
120
```

很多程序员对eof()函数的调用感到满意。然而，这会使我们的程序容易忽略输入文件格式化中的错误。

假设在文件的第三行输入50时，按了字母‘o’键而不是数字0键。当语句f>>amount;读该行时，它发现了5和‘o’，于是程序断定输入的值是5，于是将‘o’字符留在输入流中并执行下一条语句。在下一次循环中，语句f>>amount;在输入流中发现‘o’于是断定这是输入值的结束，然后就终止。下一条语句执行时，那个无助的程序就会陷入死循环。

当然，出现这种输入错误的可能性在使用键盘时比使用文件时更大，因为文件可以在执行之前进行校对，但是使用文件还是可能会出错的。有些程序员不使用运算符>>，因为它容易忽略输入格式上的错误。他们使用的是前面介绍过的函数get()和getline()，以便把数据当做字符来读取。当将输入行存放在内存中时，程序可以分析数据并在数据不正确时产生一个相应的出错消息。

容易忽略错误的另一个原因是文件结束的方式。在上面的例子中，在每个值之后都有换行字符，包括最后一个值：120。当文件的最后一个输入后跟着文件结尾的换行字符时，抽取函数>>在读取最后一个输入时停止，也就是在换行字符之前停止。在这种情况下，文件的结束只有当程序读完了最后一个输入数据时才出现。

如果没有加上最后那个换行字符，会发生什么呢？或者所有的值都写在单行而没有使用终止的换行字符，又会如何呢？如果换行字符没有跟着最后一个数据输入，抽取函数就会读取文件结束标志，于是文件的结束符就会出现。函数eof()（它在语句f>>amount;之后被调用）返回真值，因此循环终止而没有处理最后一个值。

这是不对的。程序应该以这样的方式编写：使得不论数据输入人员（或者电信软件）是否在文件的最后一个输入之后放置了换行字符，程序的处理结果都不会改变。为了解决这个问题，有些程序员完全不用eof()函数，而是使用函数fail()。

```
do {                                    // C++ or Java loop structure
  double amount;                        // local variable for input
  f >> amount;                          // get next double from file
  if (f.fail()) break;                  // stop input on no more data
  . . . . . }                           // the rest of the loop
```

当不论何种原因操作失败时，包括到达文件的结尾，函数fail()都返回true。当输入5o而不是50时，读取5，而'o'在下一次循环中在输入流中被发现。语句f>>amount;没有读取任何数据，于是fail()函数返回真。输入循环就终止。首先，早一点的循环终止比无限循环要好。其次，程序可以在循环终止后分析运行情况并产生一条关于循环是否过早终止的消息。

在第二个例子中，当值120的后面没有跟着一个换行字符时，文件的结束符就出现；但函数fail()返回false，因为语句f>>amount;正确地读取了值120。只是在下一次通过该循环，当程序想越过值120时，这个函数才返回true。因此，能正确地处理文件的最后一个值。

6.4.3 输入/输出文件对象

除了ifstream和ofstream外，C++的iostream库还定义了大量的类。对于99%的工作来说，并不需要知道这些类。这里只讨论一个fstream类，因为它把ifstream和ofstream类的特点结合在一起。

当创建ifstream和ofstream类型的变量时，并没有指定以什么模式打开它们：ifstream以缺省方式创建为输入模式，ofstream以缺省方式创建为输出模式。对于类fstream的对象，我们可以通过在创建对象时提供第二个参数来指定打开的模式。

```
fstream of("data.out",ios::out);            // output file
fstream inf("amounts.dat",ios::in);         // input file
```

其输入模式是缺省设置。其他可使用的打开模式包括ios::app（打开文件是为了把数据追加到文件尾部），ios::binary（以二进制方式打开文件，而不是以文本格式打开）等等。这些模式用二进制标志来实现。如果有必要，可以使用按位或运算符'|'把它们结合在一起。

```
fstream mystream("archive.dat",ios::in|ios::out);     // input/output
```

通常，检查一个文件操作是否成功的方法不止一个。除了上面介绍的函数fail()以外，还可以使用函数good()：

```
fstream inf("amounts.dat",ios::in);      // input file
if (!inf.good())                         // another way to do things
  { cout << "Cannot open file" << endl; return 0; }
```

我们甚至可以把文件对象当做是一个数值。当操作失败时，其值为0；当操作成功时，其值非零。下面是测试文件是否成功操作的另一个例子：

```
fstream inf("amounts.dat",ios::in);      // input file
if (!inf)                                // yet another way to do it
  { cout << "Cannot open file" << endl; return 0; }
```

可以使用同样的语法来测试读写操作是否成功。例如，通过使用带有一个单字符参数的get()函数，我们可以计算文件中的字符个数。当读操作失败（因为到达文件结束标志或其他的原因）时，它返回0，且这个值可以用来终止while循环。

```
int count = 0; char ch;
while (inf.get(ch))                  // stop when the object is no good
  count++;                           // increment count of characters
cout << "Total characters: " << count << endl;
```

一般说来，我们不必关闭文件，因为当与文件关联的文件对象在其作用域结束处被删除时，文件就被关闭了。但有时还是需要使用函数close()来显式关闭文件。

```
inf.close();                         // close the file
```

如果想在文件对象超出其作用域之前就关闭文件，就需要这样做；例如，当打开了几个文件，并且下一个文件不能够打开时。这时，在程序终止或试图去恢复之前，显式地关闭所有打开的文件是谨慎的做法。当不想让几个文件同时打开时，也可以关闭其中的一些文件；例如，我们可能会从一个文件中读取数据，然后在内存中处理这些数据，最后把结果写到另一个不久要用的文件里。

程序6-16给出了对程序6-15的一个改进。除了将结果送到屏幕显示之外，它也把结果保存在文件amounts.rep中。对I/O操作的成功与否的测试是通过将文件对象与0进行比较来实现的，这是一个常见的C++用法。输入文件在输入结束时关闭。

程序6-16　从文件中输入，显示到屏幕上并输出到文件中

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

typedef double Item;
struct Node {
  Item item;
  Node* next; } ;

int main ()
{
   int count = 0;                                  // count of amounts
   Node *data=0, *last;                            // pointers to start and end of list
   fstream inf("amounts.dat",ios::in);             // file to read data from
   if (!inf) { cout << "Cannot open file" << endl; return 0; }
do {                                               // do until end of data
  double amount;                                   // local variable for input
  inf >> amount;                                   // get next double from file
  if (!inf) break;                                 // stop input on no more data
    Node* q = new Node;                            // create new node on the heap
    if (q == 0) { cout << "Out of heap memory" << endl; break; }
```

```
    q->item = amount; q->next = NULL;          // fill node with data
    (data == 0 ? data : last->next) = q;
    last = q; count++;                         // set last, increment count
    } while (true);
  inf.close();                                 // file is not needed anymore
  fstream of("amounts.rep",ios::out);          // file to write data to
  if (!of) { cout << "Cannot open output file" << endl; }
  cout << "\nTotal of " << count << " values are loaded\n";
  of << "\nTotal of " << count << " values are loaded\n";
  if (count == 0) return 0;                    // no output if no file input
  cout << "\nNumber Amount Subtotal\n\n";      // print table header
  of << "\nNumber Amount Subtotal\n\n";                  // print table header
  cout.setf(ios::fixed); cout.precision(2);              //precision for screen
  of.setf(ios::fixed); of.precision(2);                  // precision for file
  double total = 0; int i = 0;                           // subtotal, line count
  for (Node *q = data; q != NULL; q = q->next)           // OK
    { total += q->item;                                  // accumulate total
      cout << setw(3) << i;                              // transaction number
      cout << setw(10) << q->item;                       // transaction value
      cout << setw(11) << total << endl;                 // running total
      of << setw(3) << ++i << setw(10) << q->item;       // transaction
      of << setw(11) << total << endl; }                 // running total
  Node *p = data, *r = data;
  while (p != 0)
  { p = p->next;                                         // return heap memory
    delete r; r = p; }
  return 0;
}
```

可见，在输出文件中的格式化数据语句与屏幕上的格式化数据的语句是一样的。对于图6-18中的输入数据，由程序6-16所创建的输出文件含有以下的数据。

```
Total of 4 values are loaded

Number   Amount   Subtotal

  1       330.16     330.16
  2        76.33     406.49
  3        50.00     456.49
  4       120.00     576.49
```

关于使用iostream库进行文件操作的所有内容就谈到这里。注意iostream库所含有的内容比这里讨论的要多得多，但在我们还未学习类和继承之前就详细地描述它们是不可能的。实际上，即使学习了类和继承之后，也未必要知道比这里所述的更多的内容。库iostream提供了完成同一件事的多种方法，因此不必立即学会所有的这些方法。在此，应该注意对基本语言工具的思想概念的理解。

6.5 小结

本章讨论了相当复杂的内容。讨论了C++指针的几种用途。第一种用途是用指针指向分配在栈中的一般变量，并提供一个通过别名来存取这些变量的技术。通过指针传递参数的技术将在下一章中讨论。一些程序员相信这个技术将使得他们的程序易于理解。

指针的第二种用途是在堆中而不是在栈中分配单个变量。堆变量没有名字，于是指针的

使用提供了存取它们的惟一方式。然而，用堆变量来代替一般的栈变量并没有什么实际的好处，实际中应该尽量避免这样使用。一些程序员相信这个技术减少了栈溢出的可能性。

指针的另外两种用途在C++程序中相当常见并且有用。无名字动态数组可以代替在编译时必须确定大小的有名字数组。动态数组消除了数组溢出和空间浪费的危险。它们快速并且不太复杂。在定义动态数组时，要保证没有重复地分配和释放数组。寻找空间去定义一个这样的数组以避免削弱程序的性能

指针的第二个有价值的用途是实现链式结构。它是最灵活的内存分配技术，它没有预先保留内存而是在需要时才分配。它也是最复杂的技术，因为在结点插入和删除时，有复杂的指针操作，对于遍历操作也是这样。指针运算中的错误难以发现，它们不一定总在不正确的程序中表示出来。内存的分配和释放是以碎片方式进行、每个结点都单独进行分配和释放，而不是一次几个结点同时进行；这很可能对程序性能产生负面影响。

当需要使用链式实现时，请考虑以下的一些选择：一种选择是使用动态数组，另一个合理的选择是使用标准的库模板，它提供对以下数据结构的实现：表、堆栈、队列、树等等。使用库可以把动态内存管理的灵活性和使用的简单性结合在一起。

本章的最后一个问题是处理长度没有预定义的数据序列：物理文件。说明了与键盘输入以及屏幕输出的操作相同的库对象的使用方法。使用文件可以扩展程序的存储空间，并实现数据的连续性。在将数据保存到一个文件中之后，数据就不会因为程序的终止、崩溃或电源掉电而丢失。更重要的是，数据可以在不同的时间（很可能在不同的地方）为不同的程序所利用。这大大地提高了计算机信息系统的灵活性。

在下面的章节中，我们将开始学习C++的函数和类，以及怎样创建面向对象程序。这是一个令人激动的课题！正如前面提到的，面向对象方法很可能是帮助设计人员从相对独立的片断中创建程序，并且直接在程序的代码中把程序的意图传达给维护人员的惟一方法。这个技能不可能自动获得，而必须通过学习语言的过程来积累。希望这本书的后续部分将帮助大家掌握好这个重要的技能。

第二部分　用C++进行面向对象的程序设计

本书的第二部分提供了使用C++进行面向对象程序设计的基本机制。面向对象程序设计最首要的工作是使用函数，因为对象上的每一个操作都应该用一个函数调用来实现。C++函数比较复杂，第7章将讨论程序员应该知道的全部C++函数语法。在C++中以传递参数困难而著称，因此希望本章的内容能很好地帮助大家掌握好这个重要的C++技能。

第8章继续讨论了C++函数，介绍了使用函数的方法，以及内聚性、耦合度、封装和信息隐藏的准则，并讨论了程序函数的可读性和独立性原则。此章也说明了可以通过设计客户代码调用的访问函数（而不是直接访问结构字段）来获得大多数面向对象程序设计的好处，而不一定使用C++对象。同时也说明了使用函数进行面向对象程序设计的局限性以及C++类所要达到的目标。这一章对于培养正确的面向对象程序设计的直觉非常重要。

第9章引入了C++程序设计的核心：C++类。它描述了C++类定义的语法，讨论了数据成员、成员函数、类成员的存取控制、对象初始化和析构、从函数返回对象以及其他使用对象的技术细节。这一章包含了很多复杂的细节，而且没有捷径可言，因为C++类本身就是很复杂的。但这些技术细节不应该使我们忽略使用类的主要目的：当维护人员需要理解函数之间的处理和数据流的一般意义时可以忽略微小的细节。

第10章描述了运算符函数，这是C++语法中很好的一个部分。将运算符函数引入到语言中以便支持以下的哲学思想：程序应该能够对类的对象进行任何对数值型变量可以进行的处理，比如加、减等运算。从软件工程的原则来看，这个思想并不是很重要，但它给了我们一个接触C++源代码的语法。

第11章讨论了使用C++构造函数和析构函数时潜在的危险，并介绍了怎样去认识这些危险。它也提供了几种技术去避免内存混乱和内存泄漏。这是非常重要的一章，因为一个经验不足的C++程序员可能会由于不正确地处理对象初始化而造成内存讹用。

第7章　使用C++函数编程

在前几章，我们已经学习了C++的基础知识，这些知识让我们可以实现计算机系统可能面对的各种复杂需求。

C++的内部数据类型能满足一般编程需要，我们总可以从这些基本类型中找到合乎一定范围和精确度要求的数据类型。C++的运算符让程序员把各个输入值组织成强有力而又灵活的表达式，以计算所要求的输出值。C++的控制结构将计算任务组织成适当的语句顺序，也可以在必要的时候重复计算任务，并能在条件为真或为假时改变这一计算流程。

前面学习了C++支持成分聚集的特性，而且也讨论过程序员定义数据类型，它们让程序员可以合并逻辑上属于一起的单独的数据值。合并单独的数据值让我们可以把它们作为一个整体来处理，也有助于设计人员告诉维护人员这些成分是属于一起的。我们还讨论了数组。数

组让程序员可以合并程序中进行类似处理过程的相关成分。最后，我们还讨论了C++的动态内存管理和文件管理，它们拓展了普通数组的功能和灵活性，突破了其局限性。

接下来，我们将要学习C++另一个聚集和模块化工具：函数。将单独的语句合并成函数，有助于程序员把它们作为一个逻辑单元来处理。将程序的功能分割成分开的函数是分工合作的强有力技术：不同的程序员可以并行地开发不同的函数。

在这一章中，我们来学习编写函数的技巧。这里主要强调函数的通信，即函数是如何进行数据交换的。我们将学习传递参数和从函数返回值的各种技术。这些技术的不同在于函数的实参是否被函数修改，或者是否保留了函数被调用前的值。同时，这些方法的不同也体现在：函数的参数是C++内部数据类型，还是数组，或者是程序员定义的结构（或类）。

我们还会学习与函数有关的其他技术以减少对函数名的限制，例如函数名重载和缺省参数值的使用。这些技术显著地扩大了程序员在程序实现时的选择。此外，我们还将学习如何使用内联（inline）函数消除函数调用时的实现开销。我们还将看到，当函数调用所提供的实际参数和函数头中定义的形式参数并不完全一致时，应该如何处理。

这些学习目标很有挑战性。C++函数强大而具灵活性，它让程序员在实现时有很大的选择性。我们会详细地进行阐述。

本章的内容对于掌握C++类是很重要的。不要轻易放弃，把本章的例子输入电脑，看看运行结果，通过实战会发现C++并不像想象中那么难学。

7.1　作为模块化工具的C++函数

使用C++和使用其他语言一样，程序员可以把实现某种功能算法的复杂性隐藏在相对小的模块单元中，即函数中。每个函数都是为达到某一特定目标而编写的语句集合。这些语句可以是简单语句，也可以是复杂的控制结构，或者是对其他函数的调用。这些函数既可以是编译程序自带的函数，也可以是较早前项目的专有库函数，还可以是为此特定项目定做的自定义函数。

从开发人员的角度来看，不同类型的函数之间的差别在于：为项目定做的函数的实现代码是可见的，而对于库函数，使用这些函数作为服务器函数的程序员不知道其具体的实现。程序员只是知道其接口描述：即函数调用应该提供哪些参数，函数计算的值是什么，如何从输入值计算出输出值，有哪些约束以及异常处理。

库函数的代码并不是什么商业秘密，尽管有时候它是，但通常来说是可以免费获得的。让程序员只知道函数的接口而排除函数的代码是有利的：这可以降低程序员必须处理的代码的复杂性。只有当函数可能存在需要修改的错误时，才值得去研究函数代码。当使用为特定的项目定做的程序员定义函数时，就会碰到这种情况。即使是这些函数，对函数合作性进行分析的任务也应该仅限于研究函数的接口，而不是函数的实现。

这也是我们用来评价函数通信的不同方法的准则。如果某种方法允许维护人员（或者另一个设计人员）只需要研究函数的接口，而不需要阅读函数代码，这种方法就优于需要审查函数代码的方法。

调用者（客户函数）将被调用函数（服务器函数）当成一个独立的单元来处理。在函数调用中，调用者指定函数名和实际参数（也有可能实际参数为空）。函数的调用者不知道函数（服务器）是如何工作的。客户函数只是知道服务器函数所做的工作及其接口说明。因此，使

用函数调用可以简化客户代码。通过删除具体的步骤而把它们抽象成函数调用的形式，代码可以简单明确地实现目标。

函数是模块化的最小单位，使用函数允许设计人员把大型的程序组织成更小的且更易于管理的单元。不同的函数可以指定给不同的程序员并发，以提高大型应用程序的开发速度。

如果一个算法在程序的很多地方要用到，把这个算法作为一个函数来实现，可以让设计人员在程序不同的地方调用它，而不是在客户代码中重新产生所有细节。这样做可以让目标代码更小，而且有助于代码重用。在维护阶段，短小的函数比巨大的程序更加容易理解和管理。

用作组织程序代码的模块化单元的函数也可以放到某一个库中，其他应用程序使用它们也可以增加代码重用率。

好的函数设计对于代码的可读性、程序各部分之间的依赖性以及降低应用程序的复杂性来说都至关重要。但是函数间的通信会增加程序的复杂性。在使用函数时，程序员必须在程序中三个不同部分统一代码：

- 函数声明（函数原型（function prototype）），包括函数名、返回类型和形式参数类型。
- 函数定义，包括函数头和函数体的实现。
- 函数调用，包括函数名和实际参数名（或者值）。

这三个部分必须统一。可能听起来无关紧要，因为只是三个地方而已。而且确实大多数程序员在大多数情况下都能正确处理。问题是，不管发生的机率多小，如果程序员没能正确处理，就会产生很严重的后果。

7.1.1 函数声明

C++在处理函数调用之前，必须先看到该函数的声明或者该函数的定义。因此，在函数被调用的源文件中，必须在调用函数前声明（或者定义）这个函数。正因如此，提供必要的函数原型是C++程序设计中的一个重要因素。

在函数的声明中，形式参数的类型和返回值的类型（如果有的话）必须随函数名一起描述。如果同一个函数在不同的源文件中被调用，那么这个函数必须在每一个文件中重新声明。

```
returnType functionName(type1 param1, type2 param2, ...);
```

如果函数没有返回值，那么返回类型必须声明为void，而不能什么都不写。如果没有说明函数的返回类型，编译的时候并不会出现语法错误，而是把它的返回类型当成int，而不是void。省略返回类型在C语言编程中很流行，C++为了与C语言兼容，也允许这种写法。尽管如此，这种写法仍然会令人迷惑不解，代码的维护人员需要花费更多的精力去弄清楚某一段代码到底是干了些什么。如果返回类型是int，代码应该说明返回类型是int。所以我们不赞成在C++中省略返回类型，一些C++的编译程序碰到这种情况会给出警告信息，提示这种函数的定义方法已经过时。

```
add(int x, int y);                    // int return value: bad style
void PutValues(int val, int cnt);  // no return value: void type
```

一个函数只能有一个返回值，如果客户程序需要从函数中得到多个返回值，函数可以返回结构类型变量，但这将会降低程序的执行速度。一个函数还能改变任意多个全局变量的值。这些全局变量定义在文件中所有函数之外。随着进一步的学习，我们会看到这些方法都不是好的软件工程实践方法，因为容易出错。另外，函数也能改变实际参数的值。我们将会看到，

这些方法很复杂。但不要沮丧，通过仔细地学习也是可以掌握的。

7.1.2 函数定义

在函数的定义中，其算法由C++的代码来实现。函数以函数头开始，函数头描述了其返回值类型、函数名（程序员定义的标识符）和形式参数的类型及命名列表（各形参之间用逗号隔开）。函数头的描述和函数原型之间的区别是：函数原型的结尾有分号，但函数头描述没有。另一个不同之处是形式参数的命名在函数原型中是可有可无的，但在函数头的描述中必须给出。当然，如果一个函数头中定义了在函数体中从来都没有用过的参数，这个形式参数就可以不给出命名，但是我们并不希望编出如此糟糕的程序。

函数体是有其自身作用域的语句块。在任何C++程序中，函数体中的各个语句按顺序执行，除非使用了控制结构或函数调用。

```
void PutValues(int val, int cnt)
{ cout << "Value " << val << " is found ";
  cout << cnt << " times" << endl;
  return; }                    // optional; no return value in a void function

int add (int x, int y)
{ count++;                     // global variable is modified
  return x+y;  }               // return statement and return value are mandatory
```

如果是一个返回类型为void的函数，return语句是可有可无的。它可以用在函数的任何一个地方，但不允许有返回值。执行任何return语句都会终止函数的执行，并将控制权返回给调用者。对于返回类型不是void的函数，必须至少提供一个return语句，也可以有多个return语句。每一个return语句后面必须返回函数头中指定的类型的值，（或者可以转换成return类型的类型值）。

7.1.3 函数调用

Pascal、Ada和其他流行的编程语言区分函数（function）和过程（procedure）。在这些语言中，过程是没有返回值的，但允许对形式参数或者全局变量产生副作用。在调用这个过程时，它们可以作为一个独立的语句使用，而不能作为其他表达式的一部分。这些语言中的函数有返回值而可以没有副作用。在客户代码中，函数不能作为独立语句使用，而必须放在某一个表达式中。

```
a = add(b,c) * 2;          // the use of return value in expression
PutValues(a,5);            // function call as a statement
b = PutValues(a,5)*2;      // nonsence: there is no value to return
```

C++不分函数和过程。C++的函数可以提供返回值，也可以产生副作用。函数的返回类型可以是内部数据类型或者程序员定义类型。但不允许将数组作为函数的返回值类型。如果返回值的类型是void，那么可以把该函数看成一个过程，它没有返回值，不能用在表达式中，而作为一个单独的语句使用。

与Pascal和Ada不同的是，在C++中，一个有返回值的函数也可以当做一个过程来使用，这时常常忽略返回值。也就是说，一个有返回值的函数既可以用在表达式中，也可以视为一个单独的语句。当这样一个函数被当做过程来使用的时候，其作用只在于对全局变量的副作用。

```
add(b,c);          // correct syntax even if it makes no sense
```

这样使用并不是一个好习惯。如果一个函数有返回值，客户代码应该使用它。但事实上，在C++的库函数中，相当一部分函数的非空类型的返回值都很少用到，如strcpy()和strcat()等。

花括号之间的函数体指定了函数调用时执行的操作。我们称对函数名应用了调用运算符()，调用运算符带有被逗号隔开的函数参数。

```
PutValues(17,14);          // the call operator is applied
```

大多数人都不把函数调用看做调用运算符，考虑括号中的参数表就足够了。然而，意识到函数调用应用了调用运算符很重要。此外，调用运算符可以用在不同的地方并表示不同的意义。

如果按字典排序，服务器函数的定义出现在客户函数的定义之前，编译程序就可以在编译函数调用之前看到服务器函数的定义。这种情况下，服务器函数的定义也可以作为它的声明。大多数程序员并不依赖于源代码中函数的字典排序，而是习惯使用原型。

函数的定义在一个程序中只能出现一次，但可以有多个函数原型。函数原型通常放在一个单独的工程目录的头文件中。这些文件在调用这些函数的源文件中用include宏指令导入。程序员经常会导入文件中并未使用的头文件和函数。这当然比研究到底在哪个文件中调用哪个函数要简单。对编译程序来说，这样做是可以的，因为它将忽略额外的原型。但是，维护人员不能也不应该忽略头文件。不加区别地使用原型让理解程序不同部分之间的依赖性更加困难。

C++允许我们在函数原型中不给出形式参数的命名。的确，参数名只有在函数的定义中才会用到。因此许多程序员省略掉参数名，因为编译程序不需要它们。

```
void PutValues(int, int);          // what do parameters do?
```

当参数的类型不同时，这样做就足够了，设计人员和维护人员可以很好地理解参数的角色（例如频繁使用的库函数），原型隐藏在头文件中。对程序员定义的函数而言，使用参数名可以为其角色提供帮助性的提示。

一些程序员并不在客户代码的开头处声明函数原型，而是在要调用函数的客户函数中作为文档的辅助手段进行声明。这可以清楚地告诉维护人员，就是这个函数（而不是同一个文件中的其他许多函数）使用了服务器函数。不要因为演示这个观点的例子很简单而忽视它，这是一个很重要的软件工程论题。

```
void Client(void)
{ void PutValues(int value, int count);      // list of dependencies
  int val, cnt;
  cout << "Please enter the value and its count: ";
  cin >> val >> cnt;
  PutValues(val, cnt); }
```

如果一个函数没有参数，那么在函数原型和函数的定义中可以用一个空的括号括起来，或者在括号中加入void关键字。

```
int foo();  int f(void);     // functions with no parameters
```

然而在函数调用中，只能用空括号来说明一个没有参数的函数的调用。

```
foo();  f();                 // parentheses are allowed and mandatory
```

为什么关键字void能用在函数定义和函数声明中，但不能用在函数的调用中呢？这是为了使C++编译程序实现起来简单一些，如果void能用在函数调用中，将会误导编译程序认为这是客户代码中的一个函数原型。

```
f(void);                    // this is not a call, it is a prototype
```

但是这里并没有返回值，为什么编译程序会认为这是一个函数原型呢？那是因为编译程序会认为程序员缺省了返回值的类型int，虽然这不是好的做法，但这是合法的。在第2章中曾提出过一个问题但没有马上解答，这里就是它的明确答案。

7.2　参数的提升和类型转换

C++是强类型语言，因此一个C++函数调用应对每一个函数形式参数使用正确的实际参数类型和个数。在函数体中，实际参数的值将会当做相应的形式参数的值来使用，如果实际参数的个数或顺序和相应的形式参数的个数或顺序不匹配无疑将会造成语法错误。

```
PutValues(25);              // one argument is missing: error
```

如果参数的个数和相应的顺序是正确的，但其类型和相应的参数不相容，在形式参数和实际参数之间进行匹配会导致语法错误。如果类型的值之间的转换毫无意义，就称为类型不相容的。比如说，一个类型是程序员定义类型（结构或者类），而另一个是内部数据类型数组或是另一个程序员定义类型，一个类型的值就不能代替另一个类型的值使用。

例如，我们假设a1是一个程序员定义的类型Account（结构类型），a2是一个数组（无论是何种类型），调用PutValues时将会出现两个语法错误。

```
PutValues(a1,a2);           // incompatible types: two errors
```

C++之所以对此有严格的要求，是因为PutValues()函数的代码是对整数类型的参数进行处理的。对整数而言，这些操作是合法的；对Account对象或数组而言，则是非法的。结构类型或数组类型的变量不可能具有数字的所有运算性质（如加、乘和比较运算等），虽然一个结构内部的单个元素可能会进行这些运算，但这是另一码事。

同样，我们来考虑一个画正方形的函数，它的参数类型是程序员定义类型Square。

```
void draw(Square);
```

不管Square类型如何组成或者有何属性，下面这一用法都是错误的。

```
draw(5);                    // incompatible types: syntax error
```

这同样是容易理解的，因为数字不能完成结构类型的操作（例如通过点选择运算符访问组成成分）。这一点C++和其他语言一样，十分严格和毫不妥协。

尽管如此，如果形式参数和实际参数的类型只是不匹配但并不像上所述那样不相容，就可以应用提升和隐式类型转换。这里的不匹配，指参数的类型不同，但具有相同的操作，因此类型值可以互相替代使用，并将这些类型视为相容的类型。

在进行任何计算之前，都会隐式地为一些类型执行从“较小的”数值类型到“较大的”数值类型的提升。enum类型的参数被提升为int类型，char、unsigned char和short提升为int类型。类似地，unsigned short类型提升为int类型（或者在int不比short长的机器上提升为unsigned int类型）。float类型的参数提升为double类型。这些参数提升是安全的，即不会丢失精度，也不会执行在“较小的”类型中没有定义的操作。

如果在提升后，实际参数的类型仍然不能和形式参数类型匹配，或者是参数的类型不能向高一级转化（如整数、长整数或双精度数），则使用以下隐式转换规则：任何的数字数据类型（包括unsigned）都能转换为其他数字类型。即使转换的时候会出现精度下降（如双精度小数转换为整数），也仍然可行。实际参数0能转换为任何一个数字类型或者指针类型的形式参数，但可能有损于精度。

我们再来考察一下PutValues()函数，如果传送双精度数据而不是整型数据作为参数会有什么后果呢？它们会自动地转换为整数类型。有些编译程序可能会给出警告信息，但这是合法的语句。

```
double x = 20, y = 5;          // integers are converted to double
PutValues(x,y);                // double are converted to integers
```

怎样才能告诉编译程序这种不匹配是我们想要的而不是一时大意造成的呢？一个通用的方法是使用显式类型转换将一个类型的值转换为另一个类型的值。

```
PutValues((int)x,(int)y);      // explicit cast for compiler, maintainer
```

另外一个转换方法是使用有点类似函数的语法，例如：

```
PutValues(int(x),int(y));      // alternative syntax for explicit cast
```

注意显式类型转换不仅仅是向编译程序传递设计人员的意图，更是为了我们维护起来更方便，不需要琢磨代码的意图。

如果在声明的返回类型和实际返回类型之间不匹配，也可以应用相同的转换规则。如果实际返回的类型比声明的返回类型“小一些”，实际的值就会提升为声明的类型。如果实际返回的类型不是“小一些”的，或者不能应用提升（例如int、long或者double类型的实际值），那么实际的值转换为声明的返回类型。

这些参数提升和转换情况，和C++用于表达式计算中的提升和转换情况一样。其目标都是让它们尽可能地合法。在这一点上，C++与其他现代语言相比限制要宽松一些。此外，使用继承、构造函数和重载的转换运算符（在本书稍后会讨论），也可以让C++对参数转换更加宽容。如果这些转换正是设计人员所需要的，那就很好了。但是如果设计人员犯了错误，而编译程序不会告诉程序员这个错误，那么这种情况就不好了。在任何情况下，使用隐式的类型转换都会让维护人员的工作变得困难。

比较好的做法是，准确地匹配参数和返回类型，或者使用显式的类型转换帮助维护人员理解代码的意思。

7.3 C++中函数的参数传递

C++有三种参数传递模式：按值传递、通过指针传递以及按引用传递。

当函数按值传递参数时，在函数内部对形式参数所做的修改不会影响到函数调用时使用的实际参数。当函数的参数是通过指针传递或者是按引用传递时，函数内部对形式参数所做的修改会直接影响到客户空间中实际参数的值。此外，还有传递数组参数的特殊模式。

我们将要学习不同的参数传递方式、它们的语法以及语义。我们也会尽量描述使用C++参数传递模式的指导原则，以提供最好的性能，尽可能地将设计人员的思想传达给维护人员。

7.3.1 按值调用

一个函数的实际参数可以是变量（或者符号常量）、表达式或相应类型的字面值。

```
int n = 22, cnt = 20;
PutValues(n,cnt);              // arguments as variables
PutValues(2*n,cnt-11);         // arguments as expressions
PutValues(18,14);              // arguments as literal values
```

在函数执行过程中，函数的参数被当成其作用域为函数体的局部变量处理。形式参数的名字对编译程序是可知的，它指向函数的开花括号和闭花括号之间的特定内存位置。在被调用函数的作用域外，这些局部变量名是未知的。虽然可能出于其他目的在函数外部定义了相同名字的某个变量，这个变量指向的内存位置也绝对不会与函数参数的内存位置相同。

调用函数时就定义形式参数（并分配内存空间和初始化）。为参数分配的空间来自程序的栈，并用实际参数的值来初始化形式参数。形式参数的值是实际参数值的独立副本。这些分配的空间在函数终止时被撤销（即在执行return语句或已到达函数体结尾时）。例如，考虑下面这个简单的返回参数总和的函数。

```
int add (int x, int y)         // x, y are created/initialized
{ return x+y;  }               // x, y are destroyed
```

既然形式参数（实际参数的副本）处于被调用的函数中，函数代码就可以修改它们。

```
int add (int x, int y)
{ x = x+y;       // awkward but legitimate: x is modified
  return x;      // the new value is copied into the client variable
 }               // the modified copy of argument is destroyed
```

实际参数的值不在被调用的函数作用域内，值的传递是单向的，即从调用函数到被调用函数。因此如果改变了在函数体内形式参数的值，这种改变不会传回到形式参数被撤销的调用者空间中。我们来看下面一段客户代码：

```
int a = 2, b = 3, c;    . . . .
c = add(a,b);    // variable 'a' does not change in client space
```

C++中按值调用是一种很自然的参数传递方式。在这种参数模式中，实际参数的值（变量、表达式或具体值）被拷贝到代表函数形式参数的临时变量中。此后，客户空间中的实际参数不再和这些副本有关系，函数操作副本且在函数退出时撤销副本。在函数内部对副本的修改不会影响客户的实际参数值。

这是容易理解的。当一个参数按值传递时，实际参数可以是任何的右值，如表达式、字面数值等。这些右值不能也不应该被改变。例如下面的调用中，应该保证第一个参数的值在函数调用中不被改变。

```
c = add(2*5,b);        // passing an rvalue 2*5 to a function
```

如果希望得到副作用，C++提供了按指针或者按引用传递参数的方式。这些参数传递模式比按值传递要复杂得多。如果使用不当会让程序难以维护；使用得当则可以提高程序的性能和可读性。

7.3.2　按指针调用

因为指针变量包含另一个变量的内存地址，因此称为指针，它们指向其他的程序实体。我们可以使用包含变量地址的指针来操纵程序变量，就像通过变量名操纵其相应的变量一样。在第6章中，我们学习了如何运用指针来进行动态内存管理，本节中让我们再来学习通过指针来进行参数传递的有关思想。

指针是一个强而灵活的程序设计工具，也是一个危险的工具。因此C++尽量地限制指针的使用。指针不能指向任意类型的变量。定义指针时，我们有责任决定指针将要指向int、double、Account还是Square类型的变量。这和我们在定义非指针变量时必须做的决定是一样的。

但从其他各方面来说，指针也是普通的变量。它们有类型、被赋以程序员定义的名字，它们可以被初始化，对它们也能够使用运算符。指针变量最终按照C++作用域规则被撤销在定义一个指针变量时，通过在指针的名字左边添加星号*来表示这是一个指针变量。指针被创建时也没有包含有效的值。和其他任何变量一样，指针也必须初始化或者被赋值。

```
int v1, v2;          // two integer variables; tney contain junk yet
int *p1, *p2, *p3;   // pointers to integers; they point nowhere yet
```

指针的操作包括赋值、比较以及间接引用操作。指针变量可以被赋以下面的值：

- NULL。
- 同种类型的另一个指针变量包含的值（指针可以进行自加自减整数运算）。
- 合适类型变量的地址（在其地址将赋给指针的变量名字前加C++地址运算符&即可）。

```
v1 = 123;  v2 = 456;    // variables are assigned integer values
p1 = &v1;               // pointer is assigned the address of variable v1
p2 = p1;                // pointer is assigned a value from another pointer
p3 = NULL;              // pointer is assigned the value NULL
```

符号常量NULL定义在头文件stdlib.h和其他许多库文件中，如iostream.h等。NULL与0等效。一些C++程序员喜欢使用NULL，因为它清楚地表明代码在进行指针处理。另一些程序员喜欢使用0，这两种用法都是正确的。一般说来，C程序员习惯用NULL而C++程序员喜欢使用0。

在上面的举例中，指针p1和p2都指向变量v1。指针的比较方法和其他数值类型的比较方法一样。

```
if (p1 == p2) cout << "The same address, not value\n";
if (p3 == 0) cout << "This is a null pointer\n";
if (p1 != 0) cout << "We can start working\n";
```

最后要讨论的运算符是标记为星号*的间接引用运算符。当应用于指针变量时，它标明指针所指向的值。值的类型是指针定义时所使用的类型。例如p1指向整数v1，v1的值是123因此*p1表示123，它是整数类型。在指针定义int *p1中也使用了同样的星号，这并没有用错，它表示p1是一个指向整数的指针，也表示*p1是一个整数。也正因此，星号的作用域只是一个名字。在定义整数（或者其他基本类型的变量）时，类型名应用于任何数目的变量，例如在上面那个例子中定义变量v1和v2时只使用了一个关键字int。这种做法对指针是不起作用的。例如，下面定义了一个指针和两个整数：

```
int* pt1, pt2, pt3;      // pt1 is a pointer, pt2 and pt3 - integers
```

我们再来看看间接引用的问题。间接引用的指针是它所指向变量的同义词。

```
*p1 = 42;   // v1 is not 123 anymore; it is 42
*p2 = 180;  // v1 is not 42 anymore; it is 180
*p3 = 42;   // do not dereference NULL pointers; this causes crash
```

在上面的例子中，p1指向v1。因此*p1和v1是完全等效的，除非指针被重新赋值。

```
p1 = &v2;  // p1 now points to v2, not to v1; now *p1 means 456
if (*p1 == 456) *p1 = 42;  // v2 is not 456 anymore; it is 42
```

如果间接引用的指针和它所指向的变量是同义词，那么为什么我们还需要考虑与动态内存管理无关的指针呢？答案是，我们要使用指针参数改变客户空间中实际参数的值。如果将一个指针变量传递给函数，通过使用间接引用的语法规则，该函数可以修改该指针所指向的值（即实际参数）。

例如，考虑下面这个被修改过的add()函数。和前一个版本类似，该函数计算两个参数的和。不同的是，没有返回结果，而是将结果赋值给间接引用的指针，即赋值给指针参数指向的值。

```
void add (int x, int y, int *z)    // z is a pointer to an integer
{ *z = x + y; }                    // location pointed to by pointer z is modified
```

客户代码怎样调用add()函数呢？前两个参数应该是进行加运算的整数值。第三个参数呢？还记得强类型的问题吗？它不能为double、short或者int，而应该是一个指向整数的指针。怎样才可以获得赋值给指针的值呢？它必须是NULL（本例中没有用），或者另一个指针（可能适用于其他例子，但是不适用于本例），或者是一个整数的地址——这正是我们所需要的。我们应该把希望存储总和的变量的地址作为第三个参数传递，使用&符号表示地址运算符。以下是客户代码中按指针调用时应该写的语句。

```
int a = 2, b = 3, c;
add(a, b, &c);                 // and c does change after the call!
```

本书曾提过，按值调用在C++中是一种最自然的参数传递方式，按指针传递参数也是一样的。按值传递的是指针，而不是客户空间中的值。和按值传递一样，在服务器函数中创建、初始化、使用和撤销的都是指针的局部副本。

既然传递给函数的是实际参数的地址，在函数体内访问实际参数的值时必须间接引用该指针。如果在函数内给间接引用的变量赋以新值，这种修改会持续到客户空间中。

这个逻辑看起来有些难以理解，但不要担心，很快会习惯的。要记住下面简单的检查清单，在按指针传递参数时，应该指明：

- 在调用中对实际参数使用求地址运算符&。
- 函数头中该参数为指针类型。
- 在函数体内对该参数使用间接引用运算符。

不要抵制这些逻辑，要遵循它，这样才不会出错。任何违反这个审查清单的行为都会导致错误，造成不必要的麻烦。

我们来看另一个很典型的例子：交换参数的值。如果第一个参数比第二个参数大，将交换这两个参数的值，使它们按升序排列。为了交换参数a1和a2的值，我们把第一个参数的值保存在临时变量temp中，这样就可以用a1的位置存储不同的值。接着我们把a2的值拷贝到a1中，再把存储在临时变量temp中的值移到a2中，交换数据的工作就完成了。原来在a2中的值现在在a1中，原来在a1中的值现在在a2中。程序7-1是swap()函数的实现及其客户函数main()。为了进行调试，在交换前、交换后以及调用后都添加了显示其值的语句。执行后的结果如图7-1所示。

程序7-1 副作用的传递参数（不好的版本）

```
#include <iostream>
using namespace std;

void swap (int a1, int a2)          // wrong parameter mode
{ int temp;
  if (a1 > a2)
  { cout << "Before swap: a1=" << a1 << " a2=" << a2 << endl;
    temp = a1;  a1 = a2;  a2 = temp;
    cout << "After swap:  a1=" << a1 << " a2=" << a2 << endl; } }

int main ()
{
  int x = 84, y = 42;               // values are out of order
  swap(x,y);                        // bad parameter mode; it should not work
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0;
  }
```

```
Before swap: a1=84 a2=42
After swap:  a1=42 a2=84
After call:  x=84 y=42
```

图7-1 程序7-1的输出结果

正如大家所料，在函数内正确地交换了参数的值，但是这种修改并没有持续到客户空间中，即没有交换实际参数的值。按指针传递参数应该会有帮助。下面是另一个版本的swap()函数。

```
void swap (int *a1, int *a2)              // correct parameter mode
{ int temp;
  if (a1 > a2)
  { cout << "Before swap: a1=" << a1 << ' a2=" << a2 << endl;
    temp = a1;  a1 = a2;  a2 = temp;
    cout << "After swap:  a1=" << a1 << " a2=" << a2 << endl; } }
```

这样修改后，程序好像是正确了，其实还不能得到预期的效果。temp=a1;一句是错误的，变量temp的类型是int，变量a1却不是int，而是指向int的指针。我们可以将一个整数赋值给另一个整数，也可以把一个整数指针赋值给另一个整数指针。但是不能在不同类型的变量之间赋值。不要被处理数值类型的经验误导了。不同的数值类型是相容的，它们可以互相转换。但是指针值是不相容类型，不能将它们转换为非指针类型的值。

如果看到编译程序的错误信息，不要灰心，要想一想等号右边应该怎样修改。变量a1不是整数类型，那么什么相关的变量是整数类型呢？看看函数的形式参数表，它说明了什么是整数？参数表上指明int *a1，因此，*a1才是整数，所以赋值操作应该为temp=*a1。这并不是很难，但是在函数体内进行变量修改是很费力和易于出错的事情。我们要确保对a1和a2是间接引用，而不是对temp间接引用。

```
void swap (int *a1, int *a2)                    // correct parameter mode
{ int temp;
  if (a1 > a2) {
    cout << "Before swap: *a1=" << *a1 << " *a2=" << *a2 << endl;
```

```
    temp = *a1; *a1 = *a2;   *a2 = temp;          // correct dereferencing
    cout <<"After swap:  *a1=" <<*a1 <<" *a2=" <<*a2 <<endl; } }
```

对于程序7-1中那个版本的swap()函数，编译程序还是会指出swap(x,y)的调用有问题。变量x是整数类型，但是函数swap()的参数是指向整数的指针，即整数变量的地址。程序7-2给出了修改后的正确版本，其运行结果如图7-2所示。

程序7-2　按指针传递参数（参数模式是正确的）

```
#include <iostream>
using namespace std;

void swap (int *a1, int *a2)                  // correct parameter mode
{ int temp;
  if (a1 > a2) {
    cout << "Before swap: *a1=" << *a1 << " *a2=" << *a2 << endl;
    temp = *a1; *a1 = *a2;   *a2 = temp;      // correct dereferencing
    cout << "After swap:  *a1=" <<*a1 <<" *a2=" <<*a2 << endl; } }

int main ()
{
  int x = 82, y = 42;                         // values are out of order
  swap(&x,&y);                                // correct parameter mode; it should work
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0;
  }
```

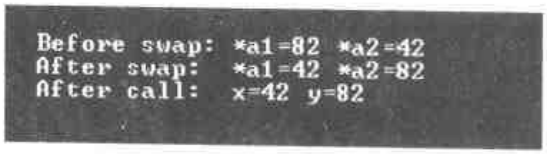


图7-2　程序7-2的输出结果

一个使用参数传递的有助记忆的规则是……等一等，我们要不要再测试一下这个程序？在函数体中有一个条件转移语句，但只是运行了一次，判断了一个条件分支。这的确是一个简单的小程序，如果花费时间测试程序的每个部分，我们可能一事无成。但是，传递参数会出现很多意想不到的情况。因此我们如下所示修改一下main()函数，只是为了确定当参数已经排序时，swap()函数无需进行交换。

```
int main ()
//{ int x = 82, y = 42;          // values are out of order
{ int x = 42, y = 84;            // values are ordered
  swap(&x,&y);                   // no swapping for ordered arguments
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0; }
```

程序的运行结果如图7-3所示。这里有两个问题。首先，有没有注意到程序的运行结果这个问题？我们常常会看到输出但是发现不了错误，因为我们没有事先把答案写出来。在本例中，即使函数swap()中有if语句，代码似乎还是不由分说地交换了参数。这可能意味着语句没有比较参数的值而是比较了其他东西。第二个问题是，有没有发现程序7-2中的代码的问题？我们没有什么方法可寻，因此发现问题更加困难。

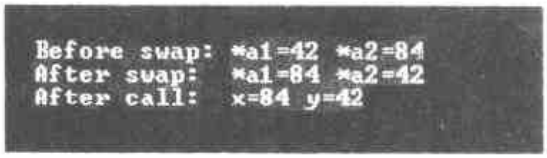


图7-3 程序7-2的输出结果（修改了main()函数）

在函数swap()中有10个星号*，但还是漏掉了两个。在比较a1和a2时，编译程序并不会给出什么错误信息，因为a1和a2的类型是一样的。如果我们想比较两个地址的大小，我们有权这样做。如果第一个变量的地址比第二个变量的地址大，则无论地址所对应的值哪个大哪个小都会进行交换操作。也就是说，编译程序不会猜测程序员的意图。正确的版本见程序7-3所示。

程序7-3 按指针传递参数（正确的间接引用）

```
#include <iostream>
using namespace std;

void swap (int *a1, int *a2)                    // correct parameter mode
{ int temp;
  if (*a1 > *a2) {                              // Oh, boy
    cout << "Before swap: *a1=" << *a1 << " *a2=" << *a2 << endl;
    temp = *a1; *a1 = *a2;  *a2 = temp;  // correct dereferencing
    cout << "After swap:  *a1=" <<*a1 <<" *a2=" <<*a2 << endl; } }

int main ()
//{ int x = 82, y = 42;                         // values are out of order
{ int x = 42, y = 84;                           // values are ordered
  swap(&x,&y);                                  // correct parameter mode; it should work
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0;
  }
```

顺便说一下，这不是一个杜撰的例子。这是现实生活中发生的实例，只不过程序的规模变小了。我们去掉了很多无关要紧的细节，突出了问题的关键所在。

现在讨论按指针传递参数的有助记忆的技巧。使用的技巧是：为参数选择名字时，应该用星号开始，而且记住在所有场合将星号作为变量名的一部份。swap()函数的第一个版本的问题（也是第二个版本的问题）就是将a1和a2当做参数名了。如果一开始就认为参数名是*a1和*a2，就很容易写出程序7-3所示的函数，即在条件语句中使用*a1>*a2。

按指针传递参数比按值传递要复杂得多，我们要注意以下几点：

- 在函数头（和函数原型中）使用指针符号*。
- 在函数体中要间接引用参数。
- 在函数体外的客户代码中使用求地址运算符&。这样做之后，对形式参数的修改会反映在客户代码的实际参数上。

每个人都会在按指针传递参数时犯错。有经验的程序员和没有经验的程序员之间的不同之处仅在于有经验的程序员犯这一类的错误比较少，且当出现错误的时候修正程序比较快。至于犯错时的生气和自我批评，……建议大家不要批判自己，毕竟这些规则不是程序员制定的。

有些程序员在函数调用时不使用取地址运算符，以使按指针传递参数更加简单。他们使用已经创建的指针指向实际参数。例如，以如下方式调用swap()函数。

```
int main ()
{ int x = 82, y = 42;            // values are out of order
  int *p1 = &x, *p2 = &y;        // set pointers to point to values
  swap(p1,p2);                   // no address-of operator
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0; }
```

这也是可行的，指针p1和p2指向的实际参数的值被正确地交换了。但是，这样做需要引入额外的程序元素即指针，并使用相同的取地址运算符设置它们。额外的操作意味着额外的出错机率以及理解代码时的额外精力。在函数调用时直接使用取地址运算符是否更加简单，这并无定论。但是如果有些程序员喜欢这样做，也没有问题。

因为可以通过按指针调用的方法在函数体内修改实际参数的值，所以只有可以通过其地址进行操作的左值才允许作为实际参数。我们不能使用右值，如表达式、具体数值或者常量等。例如，下面这个调用swap()函数的用法是错误的。

```
swap(&5, &(x+y));    // no good: no address-of operator for rvalues
```

另一个和指针传递参数有关的问题是类型转换问题。无论如何，这种类型转换都是不允许的。较早前论述的内容只适用于值的转换而不适用于指针的转换。看看下面这段代码，它试图使用swap()函数给double类型的值排序。

```
int main ()
{ double x = 82, y = 42;    // double values are out of order
  swap(&x,&y);              // no conversion from double* to int*
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0; }
```

可以使用到int*的强制类型转换，让编译程序接受这些参数。

```
swap((int*)&x, (int*)&y);   // it swaps integers, not double values
```

现在编译可以通过了，函数只是比较和交换（如果需要）了double值的整数大小范围内的部份。编译程序接受这种调用是因为我们明确表示知道自己在做什么。但是，我们所做的是错误的。在程序编译时，要确保编译程序所接受的内容是有意义的。

既然C++明确表示支持C合法的代码，按指针传递参数就是有效的C++技巧。但是，C++还添加了另外一种传递参数的模式，按引用传递，这可以消除按指针传递参数的一些缺点。我们应该尽可能少地使用按指针传递。遗憾的是，我们不可能忽视按指针传递。因为除了合法的C代码以外，还有一些按指针传递的C++库函数。动态内存管理也需要处理指针。千万不要被指针的复杂性所吓倒。

7.3.3　C++中的参数传递：按引用调用

除了指针，C++还提供了一种C语言没有的引用类型。像指针变量一样，引用变量指向另一个内存空间，包含另一个变量的内存地址。像指针一样，在定义引用变量时，要说明它所指向的变量的类型。像指针一样，我们可以定义任何类型的引用，如内部数据类型或程序员定义类型。像指针一样，引用变量也是普通的变量，可以被定义、分配内存空间、初始化、参与操作和被撤销。

但与指针类型不同的是，引用类型变量只能指向一个内存空间。这个空间的类型必须与引用本身的类型一致。引用变量不能放弃它所指向的位置，也不能指向另一个位置。正因为

如此，引用类型变量必须在声明的时候就初始化。如果不这样做，就会丧失将此引用指向某内存空间的机会，因而变成了一个无用的变量。为了表示一个变量是引用而不是指针，我们在程序员定义的变量名左边使用&符号，而不是用*。初始化引用时，不需要对引用所指向的变量应用运算符。这种标识的改变是很聪明的。这也是C++中引入引用的一个原因。

```
int v1=123,v2=456;          // integer variables; optional initialization
int *p1=&v1, *p2=&v2;       // pointers to int; initialization is optional
int &r1=v1, &r2=v2;         // references: always initialized, no operator
```

对指针而言，*p1和v1是等效的，我们需要使用间接引用运算符*。对引用操作来说，r1和v1是等效的，不需要任何运算符。这是在C++中引入引用的第二个原因。

```
if (p1 != p2) cout << "Different addresses\n";          // sure, &v1 != &v2
if (*p1 != *p2) cout << "Different values\n";           // sure, 123 != 456
if (r1 != r2) cout << "Different values\n";             // sure, 123 != 456
```

有了指针，无论是通过间接引用的指针（例如*p1）还是变量名（例如v1）来访问变量，结果都是一样的。有了引用，无论是使用无需任何运算符的引用名（例如r1）还是变量名（例v1）来访问变量，结果也是一样的。它们是同义词。

```
*p1 = 42;             // v1 (and r1) is not 123 anymore; it is 42
r1 = 180;             // v1 (and *p1) is not 42 anymore; it is 180
v1 = 42;              // r1 (and *p1) is again 42
```

这可能听起来让人迷惑：我们对指针进行“间接引用”的操作，而不是“间接指针”；而另一方面，我们无需对引用进行“间接引用”的操作。这种设计本身并没有恶意来迷惑人，只是因为历史原因。在ANSI C之前，指针常常称为引用，因为它们“引用了”变量。按指针传递也常常称为按引用传递，因此术语“间接引用”也常常用来代替“间接指针”。实际上，“间接指针”这个术语根本就不存在。

在ANSI标准化的过程中，这个术语仍然保留下来。设计C++时，需要使用一个新的术语来表示指向、参考、定向、瞄准、访问、指定、指示或者标识的意思。最后采纳了术语“引用”，因此现在我们是对指针而不是对引用进行“间接引用”操作，这是很好的。

与指针类型不同的是，引用类型在被初始化后不可能改变而指向另一个变量（的内存位置）。其原因是引用和变量永远在一起直到超出作用域或消亡时才会分开。对引用的赋值只能修改数据而不能修改数据的地址，因为引用为变量提供一个别名而已。

```
p1 = &v2;             // p1 abandons v1, points to v2 instead
p1 = p2;              // another way to do the same thing
r1 = v2;              // r1 still points to v1 that now contains 456
r1 = r2;              // another way to move data from v2 to v1
if (r1==v2) r1 = 42;  // comparison holds, and v1 becomes 42
```

到这里我们对引用类型变量有所了解，知道了它们的术语，以及用参数传递时的标识。当按引用而不是按指针传递第三个参数时，函数add()如下所示。

```
void add (int x, int y, int &z)      // z is a reference to an integer
{ z = x + y; }                       // location pointed to by z is modified
```

在上面的函数中，z是指向整数的引用类型。当函数add()被调用的时候，这个参数被分配内存，并被实际参数的地址初始化（我们会在后面讨论这是如何实现的）。对z的赋值将会修改这个间接引用指向的单元，即实际参数的值。函数体看起来好像是按值传递z似的，与

按指针传递不同的是，不需要间接引用。函数头通过引用运算符&来表示按引用传递。在函数调用时，我们必须用引用将要指向的内存地址来初始化该引用。应该怎么做呢？根据引用初始化的语法，我们直接使用变量名而不需要运算符。因此，客户代码中的函数调用如下所示：

```
int a = 2, b = 3, c;
add(a, b, c);                    // and c does change after the call!
```

因此，按引用传递参数时我们要指定：

- 函数调用时不用地址运算符的实际参数名。
- 函数头（和函数原型）中参数的引用类型。
- 函数体中无需间接引用操作的参数名。

可以看到按引用调用和按值调用很相似，在服务器函数体中无需间接引用，在客户代码中的函数调用时也无需使用取地址运算符，但由于在服务器函数头中使用引用运算符，可能导致副作用。

从某种意义来说，这种语言设计类似于Pascal语言，Pascal的关键字var和C++中引用运算符&的功能是一样的，都标识出在函数体对形式参数的修改会影响客户代码中的实际参数。而关键字可能比运算符更容易理解。

现在我们使用按引用传递实现有交换参数功能的函数。程序7-4列出了源代码。修改是很简单的，代码比使用按指针传递简单得多，出错的机率也更少。程序的运行结果如图7-4所示。

程序7-4　按引用传递参数（健壮的方法）

```
#include <iostream>
using namespace std;

void swap (int &a1, int &a2)                          // correct parameter mode
{ int temp;
  if (a1 > a2) {                                      // no dereference operator
    cout << "Before swap: a1=" << a1 << " a2=" << a2 << endl;
    temp = a1; a1 = a2;  a2 = temp;                   // no dereferencing
    cout << "After swap:  a1=" << a1 << " a2=" << a2 << endl; } }

int main ()
{ int x = 82, y = 42;                                 // values are out of order
//{ int x = 42, y = 84;                               // values are ordered
  swap(x,y);                                          // this is beautiful!
  cout << "After call:  x=" << x << " y=" << y << endl;
  return 0;
  }
```

```
Before swap: a1=82 a2=42
After swap:  a1=42 a2=82
After call:  x=42 y=82
```

图7-4　程序7-4的输出结果

我们把参数传递的规则总结在表7-1中。这里的（带有所用的运算符的）var表示变量名：在函数调用中作为实际参数，在函数头（和原型）中作为形式参数，在函数体中作为变量名字。

表7-1 关于简单变量的参数传递方式

代码元素	按值传递	按指针传递	按引用传递
函数调用	var	&var	var
函数头	var	*var	&var
函数体	var	*var	var

这张表显示出按值传递是最简单的，不需要对实际参数和函数体中的形式参数使用任何运算符。按指针传递是最复杂的，在三个代码元素——实际参数、形式参数和函数体中都需要使用运算符。按引用传递和按值传递类似，惟一的不同在于函数头中使用引用运算符。

按引用传递没有按指针传递复杂，但是也支持对客户空间中实际参数的副作用。如果形式参数在函数体内没有修改，应该使用按值传递。这有助于向维护人员说明，设计人员希望保持实际参数的值不变。

按引用传递参数并在函数体内被修改时，也有和按指针传递时同样的限制。我们只能使用左值作为实际参数，而且它们必须和形式参数完全一样的类型，因为C++不支持不同类型的引用之间的隐式转换。显式转换是可以的但却是无用的，引用变量只能访问定义时使用的类型值。使用表达式、字面值和常量都是不允许的。

提示 当调用函数无需改变C++基本类型的实际参数值时，按值传递参数。当函数需要改变其C++基本类型的实际参数值时，按引用传递参数。尽量避免通过指针传递参数。

7.3.4 结构

结构类型变量（和类对象）可以按值、按指针或者按引用传递。如果一个结构类型变量被函数用作操作的输入，而不会被函数修改，它可以按值传递。如果一个结构类型变量被函数用作操作的输出（将值传递给客户函数），即函数修改了结构的字段，就应该按指针或者按引用传递这个结构类型变量，否则这些修改不会在客户空间中生效。

前面所介绍的传递单个变量的规则同样适用于传递结构类型的变量。传递结构类型参数的其他规则和在函数体内访问结构的元素有关。

为了保持例子简单，我们来看一个简化了的类型Account。

```
struct Account {
  long num;                  // just two fields for simplicity sake
  double bal; } ;
```

函数printAccounts()接受两个Account变量作为参数，并输出其账号和余额。这些Account对象作为printAccounts()的输入变量，客户代码必须在函数调用前设置好它们的值，因为printAccounts()函数需要客户设置的Account的值来完成任务。

```
void printAccounts(Account a1, Account a2)        //server code
{ cout << "First account:  No. " << a1.num
       << "  balance " << a1.bal << endl;
  cout << "Second account: No. " << a2.num
       << "  balance " << a2.bal << endl; }
```

客户代码创建了Account对象，初始化它们的字段，然后调用服务器函数printAcco-

unts()输出有关信息。

```
Account x, y;                                    // client code
x.num = 800123456;   x.bal = 1200;
y.num = 800123123;   y.bal = 1500;
printAccounts(x,y);
```

因为讨论的重点在于参数传递，我们忽略了其他一些问题，例如，是否需要为这个简单的结构定义访问函数，或者在客户代码中直接访问结构的域更好。我们只是需要这个简单的函数例子来演示与函数之间通信相关的问题。

函数传递的基本规则在这里仍然适用，即程序员必须在以下三个地方协调代码：函数调用、函数头和函数体。根据按值传递参数的规则，在函数调用、函数头和函数体内我们都使用变量的名字而无需添加任何的运算符。当函数代码需要访问结构类型的域时，和客户代码一样使用点选择运算符。这是最简单的参数传递模式。（比较printAccounts()中的a1.bal和客户代码中的x.bal。）

下面我们再来看另外一个函数swapAccounts()，它比较两个参数的账户号码，如果不是由小到大排列，则交换两个参数。既然实际参数的值需要修改，按值传递参数就不合适了。该函数按指针传递其参数。

```
void swapAccounts (Account *a1, Account *a2)      // pass by pointer
{ Account temp;
  if (a1->num > a2->num)                          // operator
  { temp = *a1;   *a1 = *a2;   *a2 = temp; } }
```

当客户代码调用此函数时，将实际参数的地址传递给函数。

```
swapAccounts(&x,&y);
```

在这里，我们可以再一次看到按指针传递的基本规则。在函数调用时，客户代码使用取地址运算符&；在函数头中服务器代码使用星号*；在函数体中服务器代码使用间接引用运算符*。当服务器代码访问结构的域时，使用了两个字母的箭头选择运算符而不是点选择运算符。

这是一个通用的规则，而不仅限于参数传递。当左操作数是结构类型变量名时，使用点选择运算符来选择其域。当左操作数是指向结构类型变量的指针时，使用箭头选择运算符。不应该将二者混淆。程序员常常忽视这个区别。使用指针时，它指向命名的栈变量还是指向无名的堆变量，这个问题无关紧要。指针需要的是箭头选择运算符。如果在需要使用一个运算符时使用了另一个运算符，会产生错误信息（这个信息有时可能很含糊）。

一些程序员试图使用变量命名规则来提醒自己一个变量是指针而不是值。这些程序员在指针变量名前加上ptr或p。如果使用这样的命名规则，swapAccounts()函数的代码就如下所示：

```
void swapAccounts (Account *ptrA1, Account *ptrA2)
{ Account temp;
  if (ptrA1->num > ptrA2->num)
  { temp = *ptrA1;   *ptrA1 = *ptrA2;   *ptrA2 = temp; } }
```

作者比较喜欢前一种代码风格，因为个人认为（Account类型的）参数名应该是*a1和*a2（即以星号开始的参数名），而认为名字中的ptr部分容易分散人的注意力。但是这是一个通用的方法，如果参数名能够提醒程序员处理的是指针，就应该采用这样的做法。

对一些程序员来说，决定使用这种或那种选择运算符是一种额外的负担。如果先对指针间接引用，也可以对指针使用点选择运算符。使用这种技巧的swapAccounts()函数如下所示：

```
void swapAccounts (Account *a1, Account *a2)        // pass by pointer
{ Account temp;
  if ((*a1).num > (*a2).num)                        // no arrow selector
  { temp = *a1;  *a1 = *a2;  *a2 = temp; } }
```

这里的括号很重要，因为选择运算符的优先级比间接引用运算符的优先级高。没有括号的表达式，例如*a1.bal，会被编译程序理解为 *(a1.bal) 而不是 (*a1).bal，这是毫无意义的。首先，表达式a1.bal是不合法的，因为指针a1不能使用点选择运算符 其次，即使a1.num 是合法的，其域bal的类型是double，我们不能对double类型的值进行间接引用的操作，而只能对指针进行这样的操作。

使用间接引用运算符和点选择符是合法的方法，但是大多数程序员逐渐适应于从一种选择运算符转到使用另一种选择运算符，而不再努力保持运算符的一致性。如果我们总是避免使用箭头选择运算符，老板可能会怀疑我们对C++的掌握程度并没有声称的那么好。

在函数swapAccounts()中，我们使用按指针传递参数而不是按值传递，是因为实际参数的值必须修改以反映交换的结果。但是一些C++的开发人员，特别是一些有着丰富经验的C程序员，不喜欢使用按值传递参数，即使参数只是作为输入变量不会被函数修改，他们也会使用按指针传送结构类型参数。这样的程序员编写的printAccounts()函数就会是下面的样子，按指针而不是按值传送参数，使用箭头运算符访问结构类型的域。

```
void printAccounts(Account *a1, Account *a2)      // misleading
{ cout << "First account:  No. " << a1->num
       << "  balance " << a1->bal << endl;
  cout << "Second account: No. " << a2->num
       << "  balance " << a2->bal << endl; }
```

选择按指针而不是按值传递结构类型参数的原因可能是出于程序性能的考虑。当然，在例子中的Account结构很小，但是我们常常要处理每个占用成千上万个字节空间的结构对象按值传递这些结构变量会消耗运行时间和栈内存，极大地影响程序性能。有些程序员认为按指针传递参数会产生对实际参数的不授权修改或者对其数据的无意损害，尽管存在这样的危险，但是按值传递的上述不足比按指针传递的复杂性和危险性要更严重一些。按值传递参数时，即使服务器函数试图修改它们的值，这些修改也不会影响实际参数。按指针传递参数时，在服务器函数中做的修改会扩散到客户空间中。

在这里我们并不认为数据完整性是很严重的问题。毕竟，如果服务器函数试图不正确地修改它的参数，这就应该被发现和改正，而不应该以此为理由使用按值传递。

这里最主要的问题是将设计人员的意图传达给维护人员。设计人员不想修改printAccounts()函数的参数。但是设计人员没有把这种想法告诉维护人员，因为函数头清楚地表示了参数的修改是可能的——参数是按指针传递的。同样的误导信息也通过按指针传递的函数调用传达给维护人员，因为在函数调用中使用了取地址运算符。

```
printAccounts(&x,&y);        // clearly, arguments can change!
```

在本例中，查看四行代码来分析出代码的意图是不用花很多时间的。但是实际开发中可能需要查看许多行代码，而且这些代码完成的功能可能很模糊。如果按值传递参数，就没有

必要查看函数体。这是很重要的问题。

从另一个方面来说，程序的性能常常也是很重要的。按引用传递就可以鱼和熊掌兼得。我们可以避免按指针传递的复杂性，也可以避免按值传递对性能造成的破坏，同时又可以将设计人员的意图传达给维护人员。

按引用传递函数如何实现呢？程序7-5给出了例子，它实现了服务器函数printAccounts()和swapAccounts()按引用传递参数。程序7-5的运行结果如图7-5所示。

程序7-5　作为引用参数传递的结构

```
#include <iostream>
using namespace std;

struct Account {
   long num;
   double bal; } ;

void printAccounts(const Account &a1, const Account &a2)  // header
{ cout << "First account:  No. " << a1.num                 // body
       << "  balance " << a1.bal << endl;
  cout << "Second account: No. " << a2.num
       << "  balance " << a2.bal << endl; }

void swapAccounts (Account &a1, Account &a2)              // header
{ Account temp;                                           // body
  if (a1.num > a2.num)
  { temp = a1;  a1 = a2;  a2 = temp; } }

int main()
{
  Account x, y;
  x.num = 800123456;  x.bal = 1200;
  y.num = 800123123;  y.bal = 1500;
  cout << "Before swap\n";
  printAccounts(x,y);                                     // call
  swapAccounts(x,y);                                      // call
  cout << "After swap\n";
  printAccounts(x,y);
  return 0;
  }
```

```
Before swap
First account:  No. 800123456  balance 1200
Second account: No. 800123123  balance 1500
After swap
First account:  No. 800123123  balance 1500
Second account: No. 800123456  balance 1200
```

图7-5　程序7-5的输出结果

函数swapAccounts()是很简单明了的。在函数调用中，使用了结构变量的名字，在函数头使用了引用标识，而在函数体中使用了结构变量的名字。在函数体中用点选择运算符访问该参数的域，这就无需考虑应该选择哪个正确的选择运算符了。可以看到，按引用传递

结构类型参数比按指针传递参数的表达方式简单。

函数printAccounts()也很简单。和按值传递参数一样，它的函数调用使用了结构变量的名字，在函数体中使用了参数名并访问它们的域。函数头中的两点不同是：在参数名字前使用了引用标识，在参数类型前面加上了关键字const。第一点不同是避免了拷贝实际参数，因为传给函数的是域的地址而不是副本。第二点不同是防止了在函数中修改参数的值，并清楚地告诉维护人员没有修改参数。这就是一个保证，不需要再查看函数体以确认。

const修饰符的用法和定义变量时的用法相似。这个修饰符可以用在值、指针和引用上。用在值上意味着我们不能通过直接的赋值或指针或引用来改变被修饰的值。

```
const int val = 10;          // initialization is mandatory
val = 20;                    // syntax error: assignment is not allowed
int *p = &val;               // illegal so as to prevent indirect change *p = 20;
int &r = val;                // illegal so as to prevent indirect change r = 20;
```

当对指针或者引用使用const修饰符时，根据const的位置可有两种不同的含义，如果用在类型名前，则表示指针或者引用所指向的值不能通过间接引用指针或者通过引用来改变。

```
const int val = 10;
const int *constp = &val;    // OK, but *constp=20 is a syntax error
const int &constr = val;     // OK, but constr=20 is a syntax error
```

注意直接改变变量val的值是不合法的，例如val=20，因为val已声明为常量。间接的修改在这里也是不合法的，这并不是因为变量val是常量，而是因为指针（和引用）变量定义为指向常量。对于指向常量的指针（或者引用），间接的修改是不合法的，即使它指向一个非常量的值。一定要注意区分这些差别。

```
int value = 10;               // this variable can be modified
const int *constp = &value;   // *constp=20 is still a syntax error
const int &constr = value;    // constr=20 is still a syntax error
```

当const修饰符在类型名和指针名之间会出现什么情况呢？这意味着指针是常量。它可以被间接引用，它所指向的数据的值也可以改变，但是指针不能再重新定向以指向另一个变量。虽然这类指针没有硬性规定要初始化，但是初始化是必需的。如果常量指针在定义时没有初始化，就会毫无意义，因为以后也不能赋值了。

```
int value = 10;                 // it is not const in this example
int* const pconst = &value;     // they are married for life
*pconst = 20;                   // this is OK, value is not const
pconst = NULL;                  // syntax error: pointer is constant
```

不能声明一个引用是常量，因为在C++中所有的引用都缺省为常量。它们在定义时初始化，并且不能指向其他的位置。在参数传递时使用的标识不仅表明引用不能指向另一个位置，也表明引用指向的位置的值不能修改。在函数调用时，这个引用被实际参数的值初始化。

如果设计人员厌倦了分析所有这些细节，按引用传递了结构但是没有使用const修饰符，会怎么样呢？没有问题。下面是简化了的printAccounts()函数。

```
void printAccounts(Account &a1, Account &a2)     // can they change?
{ cout << "First account:  No. " << a1.num
       << "  balance " << a1.bal << endl;
  cout << "Second account: No. " << a2.num
       << "  balance " << a2.bal << endl; }
```

这个函数代码编写正确吗？答案是肯定的。即使该函数不作修改其参数的承诺，它也不会修改参数。这显然也是对软件危机的一点贡献。设计人员没有告诉维护人员他在设计时的想法，即设计人员并不想在函数体内修改参数的值。

一些程序员认为使用const修饰符很有用，因为可以防止函数不授权地改变参数的值。但这不是最主要的问题。并不是每次调用函数都会发生错误。但是，每次查看函数代码时，我们都想知道对参数进行了什么操作，使用const修饰符是一种确定的方式来说明参数只是作为输入而已。类似地，不使用const修饰符应该是表明函数要修改参数，而不是表明程序员厌倦了分析各种细节。

一定要切记这个使用规则：如果我们写的函数修改了一个结构参数，则按引用传递它们，不使用const；如果结构参数未被更改，则按引用传递并使用const。

同样，对维护人员来说，没有使用const修饰符意在告诉他在函数体内会修改参数的值，而不是告诉他开发人员漫不经心。这听起来有些烦人，的确这样，但C++中没有其他更好的办法区分一个引用参数是作为输入还是输出。

提示　为了避免对性能造成破坏，应该尽量避免按值传递结构类型；为了避免不必要的复杂性，要尽量避免通过指针传递结构类型。总是按引用传递结构类型。如果函数不会更改参数，使用const明确指出这一点；如果函数会修改参数，则不使用const。

用按引用传递结构类型变量比按值或者按指针传递有许多优势，快速（不需要拷贝）而且简单（不需要在调用时使用取地址运算符，也不需要在函数体内进行间接引用）。如果使用得当，按引用传递可以将设计人员的意图传达给维护人员。在C++中按引用传递很普遍。请正确使用它。

7.3.5　数组

数组常常以一种与按指针传递类似的特殊模式传递。虽然其表示符号和按指针传递类似，但并不完全一样。如果在服务器函数体内改变了数组的某些元素的值，这些改变在客户空间中的实际参数数组中也是可见的。

这是C++中将数组作为参数传递的惟一模式。无论作为输入参数还是输出参数都必须使用这种模式。与参数传递的其他情况类似，我们也必须在函数调用、函数头和函数体中保持代码一致。

这里有一个函数例子。其功能是把第二个数组参数内容复制到第一个数组参数中。由于函数不知道数组的大小，第三个参数必须指定被复制的元素的个数。

```
void Copy(double dest[], double src[], int size)
  { for (int i=0;  i < size;  i++)            // classic array loop
    dest[i] = src[i]; }
```

当然，在调用这个函数时必须确保数组有足够的元素完成操作。C++没有给程序员提供内存崩溃的保护措施。下面是客户代码的例子。

```
double x[100], y[100];  int n=0;
do {
  cout << "Enter data value (0 to terminate): ";
  cin >> y[n++];                // fill array y[], assign n
```

```
    . . . } while (true);
Copy(x,y,n);                    // copy n components of y[] into x[]
```

正如该例子所演示的，把数组作为参数传递时要指定：

- 函数调用中不带中括号的数组名。
- 函数头中数组名后空中括号。
- 函数体中的数组分量（或者不带中括号的数组名）。

与按指针传递参数不同，客户进行函数调用时不必使用取地址运算符&，在函数头也不加指针标识符。如果想强调数组参数和指针参数之间的相似点，也可以用以下方式编写函数。

```
void Copy(double *dest, double *src, int size)
  { for (int i=0;  i < size;  i++)        // classic array loop
    dest[i] = src[i]; }
```

也可以用与按指针传递时类似（但不一样）的方法间接引用每个数组元素的地址。

```
void Copy(double *dest, double *src, int size)
  { for (int i=0;  i < size;  i++)        // classic array loop
    *(dest+i) = *(src+i); }               // or *dest++ = *src++;
```

如上的使用方法已经和按指针传递的形式很接近了，函数头也使用了指针表示符。不同之处是，函数体内不是对参数进行间接引用，而是对参数和下标的和进行间接引用。如果希望强调数组和指针的相似性，也可以在函数调用时使用数组的第一个元素的地址。

```
double x[100], y[100];  int n;   // fill array y[], assign n
Copy(&x[0],&y[0],n);             // copy n components of y[] into x[]
```

上面的调用例子中，使用的是数组第一个元素的地址，而不是实际参数的地址。

无论我们使用何种形式的句法，都不能在函数调用或者在函数头中区分出数组参数的角色是输入参数还是输出参数。在本例中，数组src[]仅作为输入参数，它的元素的值被函数用来完成工作而不会作为调用的结果被修改。数组dest[]作为输出参数，不论在调用前其元素的值是什么，都不会被函数使用，而它的元素的内容会作为调用的结果被修改。但是，这两个数组的标识在三个地方（函数调用、函数头和函数体中）都是一模一样的。这并不合理。当代码的设计人员不希望在函数中修改数组时，好的做法是使用const修饰符来告诉维护人员。

```
void Copy(double dest[], const double src[], int size)
  { for (int i=0;  i < size;  i++)
    dest[i] = src[i]; }      // src[i] = dest[i]; is a syntax error
```

与传递结构类型参数类似，坚持在传递数组参数时在必要的地方加上const修饰符是极其重要的。它可以阻止函数改变输入变量，更重要的是，它向代码的维护人员清楚地表明了代码的意图。

如果参数标记为const，则它可以接受的实际参数既可以有const修饰，也可以没有const修饰。这样做被认为是安全的，一个没有标记为const的参数是可以被修改的，但函数体并没有改变它，这毫无问题。但是，把一个标记为const变量作为没有const修饰符的函数的参数时就会出现错误。

```
const double c[] = { 1.1, 1.2, 1.3, 1.3 } ;
Copy(x,c,4); Copy(y,a,4);   // ok: c[] and src[] are const arrays
Copy(c,x,4);                // syntax error: c[] is a const array, dest[] is not
```

有时候，const修饰符的使用让设计人员倍感麻烦，但不要放弃使用它。我们曾说过，高级语言在编写代码时的复杂性是为了使其可读性更好。一定要保证操作和它所表明的意图是一致的，也一定要确保把参数传递给服务器函数时保持一致。

下面是一个有点难度的例子。这是我以前编写的简单函数，用于计算数组元素的总和，数组元素的个数是给定的。

```
double sum (double a[], int n)
{ double total = 0.0;                    // initialize the tally
  for (int i = 0;  i < n;  i++)          // another classic loop
    total += a[i];                       // accumulate total
  return total; }
```

后来，我需要计算有效数组元素的平均数。为此，我必须首先算出总和，再除以元素个数。即使从头开始做也不会很困难，但是既然已经编写了求和函数sum()，我决定使用它。

```
double avg (const double a[], int n)
  { return sum(a,n)/n; }                 // syntax error
```

这是一个将参数继续传递给另一个服务器函数sum()的例子。在sum()函数中，我们演示了读取数组元素的方法，即在数组名后加上一个用中括号括起来的元素下标。在函数avg()中我们演示了在函数体中使用数组名而无需中括号的方法。

但本例的主要问题是代码不能通过编译。编译程序的逻辑是：在avg()的函数头中数组a[]声明为const，因此在avg()内不能改变。然而avg()的函数体将a[]作为参数传递给sum()函数，而sum()并没有保证它的参数是保持不变的，即有可能改变其参数，这就和avg()做的许诺相违背了。

编译程序并不去理会sum()函数是否是真的改变了数组元素的值。还记得那个故事吗？如果没有铜线，就可以证明使用的是蜂窝电话。照此逻辑，编译程序将把对sum()的函数调用标记为语法错误。

这听起来好像有些武断，如果编译程序检查一下sum()函数到底对其参数进行了什么操作就好了。但是要考虑到，函数sum()可能放在不同的文件中，编译程序只知道它的原型，而且即使函数放在和avg()相同的文件中，编译程序也不会费力去分析程序中值的流程。

为了改正这些情况，必须做到恰到好处，将所有不会被修改的参数定义为const。代码的意图必须在服务器函数的接口中反映出来

```
double sum (const double a[], int n)       // array is input
  { double total = 0.0;
    for (int i=0;  i<n;  i++)
      total += a[i];
    return total; }
```

一个有数组类型参数的函数原型与其他函数声明的规则一样：函数头以分号结束。

```
double sum (const double a[], int n);      // function prototype
```

在函数原型中，参数名是可有可无的。但像数组这样特殊的参数如何表示呢？如果省掉参数，看起来会有点怪，但是编译程序能够理解。

```
double sum (const double[], int);      // parameter names are optional
```

如果使用类似指针的标识传递数组，函数原型同下述形式类似，

```
double sum (const double*, int);       // no change to value pointed to
```

提示 在C++中，传递数组参数的方式只有一种。为了区别用来输入和输出的数组参数，在用于输入的数组参数（它们不会被函数修改）前加上const修饰符。

在C++中将数组作为参数传递是高效的。不需要复制数组数据，因此节省运行时间和堆栈空间。与将结构类型变量作为参数传递相似，一定要明确：如果没有使用const修饰符就意味着函数体内会改变数组元素的值。这个约定会有助于我们缓解软件危机。

7.3.6 类型转换的进一步讨论

正如在7.2节中所论述的那样，C++对参数传递有很严格的规定：如果函数需要一个标量值作为参数，就不能使用结构类型或者数组作为实际参数。

这个规则可以扩展到结构类型中，如果函数需要指定类型的结构（或者类）作为参数，就不能使用标量值、数组或者不同类型的结构作为实际参数。那样做一定会导致语法错误即使不同类型的结构和函数期望的结构的元素完全一样，也无济于事。即使两种类型的域排列、类型和名称都一样，也仍然不行。因为编译程序只会接受其类型名和形式参数的类型名相同的实际参数，而不会进行任何其他的分析。

按值传递结构完全遵循以上规则。按指针传递结构或按引用传递结构或传递数组就有些不同。

在关于将结构作为参数传递的一节中，我们讨论了结构类型Account和按指针传递Account参数的函数swapAccounts()。

```
struct Account {
  long num;                    // just two fields for simplicity sake
  double bal; } ;

void swapAccounts (Account *a1, Account *a2)     //account is needed
{ Account temp;
  if ((*a1).num > (*a2).num)
  { temp = *a1;  *a1 = *a2;  *a2 = temp; } }
```

下面我们考虑另一个结构类型Transaction，并尝试将这个类型的变量传递给swapAcccunts()函数。结果编译程序显示语法错误。

```
struct Transaction {
  long num;                    // same name and the same type as in Account
  double amt; } ;              // different name but the same type

Transaction tran1, tran2;    ...   // client code
swapAccounts(&tran1,&tran2);       // error: wrong argument type
```

但是它们的确是一样的结构，而且我们希望使用swapAccounts()来交换Transaction类型变量而不是重新编写一个函数swapTransactions()。我们很清楚地知道自己在干些什么，我们也想让编译程序接受这个代码而不是标记为错误。C++提供了告诉编译程序我们知道自己在干什么的方法，即强制类型转换或显式类型转换。要做的只是把指向Transaction类型的指针强制转换为指向Account类型的指针，这样编译程序就能接受代码了。

```
swapAccounts((Account*)&tran1,(Account*)&tran2);    // no syntax error
```

这种用法虽然极其怪异，但是有效，它维护了我们告诉编译程序知道自己想干什么的权

力。但一些程序员认为这样的代码是“搬起石头砸自己的脚”，自找麻烦。的确，在维护过程中Account和Transaction的类型都可能变化，后果将不堪设想。还是编写一个小小的swapTransactions()函数可能会更加安全。

按引用传递结构类型变量也有同样的问题，在程序7-5中，我们讨论了函数printAccounts()，该函数期望按引用传递Account参数。如果我们传送Transaction类型变量作为实际参数，编译程序把代码标记为语法错误。

```
Transaction tran1, tran2;  ...  // transaction objects
printAccounts(tran1,tran2);     // syntax error: wrong argument type
```

如果我们坚持要这样做，我们也可以将Transaction类型的引用强制转换为Account类型引用，从而告诉编译程序我们知道自己在做什么。

```
printAccounts((Account&)tran1,(Account&)tran2);   // no syntax error
```

这样的用法也不符合软件工程思想，但C++允许我们这样做。注意，强制类型转换清楚地告诉维护人员在设计代码时我们的思路（即为一个新的目的重用了已存在的函数，而不是写一个新的函数）。

数组变量的情况也差不多，没有修饰符的数组名和指向第一个数组元素的指针等价。

如果函数以某一特定类型的数组作为其形式参数，而我们使用一个标量变量、结构类型变量或者不同类型的数组作为实际参数，就会导致语法错误。因为C++是强类型语言。

例如copyAccounts()函数，它把一个Account类型的数组复制到另一个数组中。

```
void copyAccounts(Account dest[], const Account src[], int size)
  { for (int i=0;  i < size;  i++)
    dest[i] = src[i]; }                    // same code as for Copy()
```

如果我们试图使用这个函数复制一个Transaction类型对象的数组或者一个整数数组，编译程序会产生语法错误。

```
Transaction tran1[5], tran2[5];   ...      // transaction arrays
int data1[20], data2[20];   ...            // arrays of integers
copyAccounts(tran1,tran2,5);    // syntax error: wrong argument type
copyAccounts(data1,data2,20);   // syntax error: wrong argument type
```

使用强制类型转换可以让编译程序接受这样的函数调用。

```
copyAccounts((Account*)tran1,(Account*)tran2,5);       // no error
copyAccounts((Account*)data1, (Account*)data2,20);     // no error?
```

因为Transaction类型的变量和Account类型的对象有相同的大小，所以第一个函数调用是合理的。第二个函数调用就不合理，它会复制20块大小为Account的内存，而不是复制20块int大小的内存，从而导致计算机内存崩溃。

传递内部标量类型数组的情况也一样。前一节中的函数Copy()期望double类型的数组作为参数。如果传递int类型的数组作为参数，编译程序会产生错误信息。如果将实际参数强制转换为(double*)，编译程序产生的代码会复制double大小的内存块而不是int大小的内存块。

幸运的是，我们不会因为不小心而碰到这种情况，因为需要进行强制类型转换才会迫使编译程序接受这样的代码。

7.3.7 从函数返回值

在前面的例子中，所有函数的返回类型不是void就是像int那样内部标量数据类型。C++函数是按值返回结果的。这就意味着，函数空间中的返回值先经过复制，再将副本赋值给调用方空间中的变量。

如果函数的返回类型被定义为某种结构类型，C++允许函数返回一个结构。在下面的例中，我们将使用一个修改后的swapAccounts()函数版本。在前面的版本中，它将两个Account类型的参数的账号进行比较，如果不是由小到大排列，则交换两个参数（即第一个参数的num域值较第二个参数的num域值大）。与前面版本不同的是，现在这个版本返回的两个Account类型参数中num域值较大的那一个参数变量（参数a2）。

```
Account swapAccounts (Account &a1, Account &a2)  // new return type
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2;  a2 = temp; }
  return a2.num; }                  // bad return type: no conversion from long
```

这里又应用了强类型规则：如果其返回类型定义为结构类型（如这里是Account），则在函数体内的返回表达式和调用方空间中的变量都必须使用相同的结构类型。既不能是表达式，也不能是调用方的内部数据类型，另一种结构类型或者数组类型变量。这些类型之间的转换是不合法的。

```
Account ac1,ac2,ac3;  long acc_num; ...   // value in caller space
acc_num = swapAccounts(ac1,ac2);          // error: no conversion
```

与函数的参数传递类似，函数的返回值也需要三个地方的代码保持协调一致：a) 返回类型，b)返回表达式，c)调用方空间中的变量。

```
Account swapAccounts (Account &a1, Account &a2)
{ Account temp = a1;              // initialize temp to a1
  if (a1.num > a2.num)            // check if numbers are out of order
  { a1 = a2;  a2 = temp; }        // swap arguments if out of order
  return a2; }                    // correct type of return expression

ac3 = swapAccounts(ac1,ac2);      // correct use of return value
```

上面的函数用法虽然正确，但在传送很大的结构类型变量时，程序的效率将会变得很低(特别是当函数被频繁调用时)。因此很多函数的返回值都只是设计为void或int类型，也可以是表示函数调用成功还是失败的布尔值。

C++不允许使用数组作为函数的返回值类型，但却允许返回指针或者引用类型。这样可以消除复制函数的返回值的问题。下面的例子是函数swapAccounts()，它比较两个Account参数的num域，如果它们不是由小到大排列，则交换其相应的实际参数，并返回指向num域值较大的Account变量的指针。

```
Account* swapAccounts (Account &a1, Account &a2)  // return pointer
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2;  a2 = temp; }
  return &a2; }             // return the address of actual argument
```

再次强调，所有这些类型必须在三个地方保持相容：a)被声明的函数返回类型，b)函数的

返回值表达式的类型，c)调用方空间中的变量类型。

```
Account ac1, ac2, ac3, *ac4;          ...
ac4 = swapAccounts(ac1,ac2);  // ac4 is a pointer, not an Account
ac4->num = 0;                 // it affects ac1 or ac2 that are not mentioned here
*ac4 = ac3;                   // copying ac3 into structure with larger number
```

可以看到，返回指针允许我们在客户代码中使用相当难以理解的编程模式，例如ac4->num = 0。这将会导致ac1或者ac2的值的改变，因此维护人员必须查看服务器函数，例如swapAccounts()的代码，才知道这段代码做了些什么。而服务器函数也可能不十分明了，需要查看代码的其他部分，这就增加了维护的复杂性，增加了错误的机率。返回指针甚至允许我们在客户的内存空间中使用更加糟糕的语法，例如使用下面的代码设置num域值较大的为0。

```
swapAccounts(ac1,ac2)->num = 0;     // is not this nice?
```

如果我们想把变量ac3拷贝到具有更大余额的结构中，我们可以用如下的方法：

```
*swapAccounts(ac1,ac2) = ac3;       // actually, this is not very nice
```

这些代码是正确的，但是没有很好地传达设计人员的意图。维护人员必须花费额外的时间来掌握代码的意思。如果不希望使用这种编程模式，swapAccounts()的设计人员可以通过将返回值定义为指向const对象的指针来表达这种意思。

```
const Account* swapAccounts (Account &a1, Account &a2)  // new idea
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2;  a2 = temp; }
  return &a2; }             // return the address of actual argument
```

这就意味着返回地址不能用来修改它所指向的值。如上所示定义swapAccounts()后，下面的代码就是有语法错误的。

```
*swapAccounts(ac1,ac2)=ac3;           // error: no changes to a const object
swapAccounts(ac1,ac2)->num = 0;       // error: no change to a const
```

这样的返回值的使用情况就很受限制。它不能赋值给正确类型的任意指针，因为这个指针有可能改变它所指向的值。

```
Account *ac5 = swapAccounts(ac1,ac2);     // syntax error
ac5->num = 0;                             // hence this code will never compile
```

这种返回值只能用来访问对象的成员、或者赋值给指向const对象的指针。

```
const Account *ac5 = swapAccounts(ac1,ac2); // this is OK now
ac5->num = 0;                               // this code still does not compile
```

当我们使用指针作为函数的返回值时要特别注意，在调用方的内存空间中存在的被返回的指针在服务器函数终止后是否还有意义。因此，返回指向只在服务器函数作用域定义的变量的指针并不是一个好的习惯。在前面的例子中，返回的指向函数形式参数的指针实际上是指向实际参数的指针，它在函数调用后仍然在客户内存空间中存在。但是情况并不总是这样。例如，下面的swapAccounts()，它所返回的指针指向的结构只是存储了参数a1的num值。

```
Account* swapAccounts (Account &a1, Account &a2)   // return pointer
{ Account temp = a1;              // temp holds data from a1
  if (a1.num > a2.num)
```

```
  { a1=a2; a2=temp; }                    // a1 might change, but temp holds its data
  return &temp; }                        // whose address is this, anyway?
```

当函数执行到其作用域的闭花括号时，temp被撤销。因此，客户代码中的指针ac4并不是指向一个保存了ac1变量的数据的结构，而是指向不再属于该程序的内存位置。这种情况我们称之为“悬垂指针”（dangling pointer），表示一个指向已经消失的对象的指针。

并不是所有的运行环境都足够成熟，能够捕获这种内存访问混乱，但是有些运行环境可以做到。而且，temp使用的空间可能在有些时候没有被其他目的所使用，使用它的地址的客户代码可能会产生正确的结果。

于是我们再次面对这种情况：一段代码通过了编译，执行程序的每一个分支都获得了正确的运行时结果，但是这仍然不能足以保证该程序是正确的。

返回指向局部变量的指针是不安全的。比较安全的是返回指向堆内存的指针、或者指向客户空间中的变量的指针。下面的例子返回了指向客户内存空间中的变量的指针，解决了指针悬垂的问题。

```
Account* swapAccounts (Account &a1, Account &a2)   // return pointer
{ Account temp = a1;                    // temp holds data from a1
  if (a1.num > a2.num)
  { a1=a2; a2=temp;
    return &a2; }                       // data from a1 is now in a2
  return &a1; }                         // data from a1 remains in a1
```

如果函数返回的指针指向客户内存空间的变量，我们必须清楚该变量是否是常量。考虑下面的例子函数，它比较两个Account变量的bal域，并返回指向bal值较大的Account对象的指针。

```
Account* largerBalance (const Account &a1, const Account &a2)      //no!
{ return (a1.bal>a2.bal) ? &a1:&a2; }      // pointer to actual argument
```

这是一个说明常量对象的好例子。函数没有改变参数的状态，而只是把它们当成计算的输入变量使用。因此我们在函数头中使用const修饰符。但是代码竟不能通过编译，为什么呢？因为返回的指针并没有定义为指向常量对象的指针。函数允诺不会修改实际参数的状态，但是它返回的指针指向了一个实际参数，从而可以修改该参数的域。编译程序将该段代码标志为语法错误，可以防止我们写出可能通过编译但是实际上修改了常量对象的客户代码。

```
const Account acc1 = {325,1000.0}, acc2 = {370,100.0};   // immutable
Account *p = largerBalance(acc1,acc2);   // valid syntax but dangerous
p->bal = 0;            // valid syntax but modifies a constant object
```

这样看起来似乎编译程序太苛刻了。即使largerBalance()函数的参数定义为const，但是还是应该可以把非常量的变量作为参数传递。

```
Account acc1 = {325,1000.0}, acc2 = {370,100.0};   // mutable objects
Account *p = largerBalance(acc1,acc2);     // valid syntax but dangerous
p->bal = 0;         // OK for non-const but not OK for const objects
```

但编译程序并没有那么聪明，它不能自动判断出传递给参数的对象是常量还是非常量。正如我们处理其他事物的方法一样，干脆禁止所有相关的行为。这里，C++要求在返回类型使用const修饰符。（注意，我们并不建议从函数头中删除const修饰符。）

```
const Account* largerBalance (const Account &a1, const Account &a2)
```

```
{ return (a1.bal>a2.bal) ? &a1 : &a2; }        // this code compiles
```

上面这段代码可以通过编译。它又是如何阻止客户代码改变常量对象的呢？很简单，下面这段代码就不能通过编译。

```
const Account acc1 = (325,1000.0), acc2 = (370,100.0);     // immutable
Account *p = largerBalance(acc1,acc2);    // now this is a syntax error
p->bal = 0;      // no syntax error, but compiler wants to prevent this
```

编译程序希望阻止p->bal = 0这样的赋值。既然在语法上正确的操作，编译程序就将给指针p的赋值标志为语法错误，因为给p的赋值并没有允诺不修改p所指向的对象。编译程序要求我们采取措施保持代码一致，即将p定义为指向常量对象的指针。

```
const Account acc1 = (325,1000.0), acc2 = (370,100.0);  // immutable
const Account *p = largerBalance(acc1,acc2);  // OK: no syntax error
p->bal = 0;                                   // now this is a syntax error!
```

在Account对象没有定义为不可修改的情况下，这样的要求仍然是有些苛刻的，指针p还是不能用来修改它们。但是为了常量对象的安全（也因为编译程序的设计人员不愿意进行程序的数据流分析），这是我们必须付出的代价。

函数返回地址的最安全用法是用于动态内存管理。服务器函数在堆内存中分配空间，然后把指向这个空间的指针返回给客户代码使用。（其他一些函数应该在客户使用后删除这个堆内存。）下面的例子中，allocateAccounts()函数为Account对象的动态数组分配了空间，数组的大小作为实际参数传递。

```
Account* allocateAccounts(int size)     // pointer to non-const
  { if (size <= 0) return 0;            // test argument validity
    Account *p = new Account[size];
    if (p == 0)                         // simple but too crude
      cout << "Out of memory in allocateAccounts()\n";
    return p; }                         // NULL if anything went wrong
```

如果出错则函数返回NULL。检查内存分配是否成功是客户代码的责任。

返回引用是另外一个避免在运行时复制结构值的方法。从概念上说，它和返回指向结构的指针类似。而实际上两者大不相同，因为缺省情况下C++中的引用就是常量。我们来看下面这个版本的swapAccounts()函数，它返回的是具有最大账号的实际参数的引用。

```
Account& swapAccounts (Account &a1, Account &a2)
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2;  a2 = temp; }
  return a2; }                          // wrong type if return &a2;
```

注意到它和按值返回结构的代码看起来几乎完全一样。惟一的不同之处是在返回类型的后面加上了&运算符。这样，如果返回&a2而不是a2就不正确。因为a2是Account类型，用来初始化一个Account引用；而&a2是一个指向Account类型的指针，不能用来初始化一个Account引用。在C++中这两种类型是不相容的。

但是，该客户代码容易出问题。

```
Account ac1,ac2,a3, &ac4; ...       // this time around, it is reference
ac4 = swapAccounts(ac1,ac2);        // this is a pipe dream, not real code
```

这段代码是不正确的。它企图把swapAccounts()的返回值赋给引用变量ac4，但这

时候已经太迟了，因为引用类型变量必须在声明的时候初始化，而上面这段代码没有这样做。将返回值进行赋值的惟一用法是用来初始化。

```
Account ac1,ac2,a3; ...
Account &ac4 = swapAccounts(ac1,ac2);       // this time it is OK
ac4.num = 0;                    // it affects ac1 or ac2 that are not mentioned
ac4 = a3;                       // copying a3 into structure with larger number
```

因为ac4是等效于ac1或者ac2的，所以这段代码的功能就有点不清晰。既然swap-Accounts()函数的返回值是一个引用，我们可以用下面一个奇特的语法，它是合法的：

```
largerBalance(ac1,ac2).num = 0; // is not this nice?
largerBalance(ac1,ac2) = a3;    // actually, this is not nice at all
```

在其他所有的语言中，这种怪异的用法是不允许的，包括一般的C语言在内，即函数的返回值不能当成左值使用。但是这在C++中是允许的。用不需要这种计算类型的语句重新编写算法可能更好。

在前面的例子中，我们小心地把ac4定义为引用类型而不是一个结构类型变量。但如果我们使用一个结构类型变量接收从函数按引用返回的值，则会发生复制数据的操作，就和从函数按值返回结果时一样。

```
ac3 = largerBalance(ac1,ac2);    // ac3 is an Account, not a reference
```

因为ac3是一个普通的结构类型对象，所有按引用返回的好处将体现不出来。

这些讨论十分复杂，涉及许多变化的与复杂关系相关的思想。阅读和理解返回结构的值、指针和引用的代码毕竟不是小问题。为了便利和性能而大费周折，值得吗？有其他简单的方法达到同样的结果吗？

如果C++函数只是返回逻辑标志以表示函数调用的成功或者失败，这可能是个好主意。但是，有时候，特别是涉及动态内存管理时，返回指针会更有意义。每次需要返回指针或者引用时，一定要考虑以下的两点：a)是否确实得到性能上的好处？b)是否会破坏程序的完整性？

7.4 内联函数

与使用C++函数进行程序模块化相关的另一个有用技巧是使用内联函数。在前面的学习中我们知道，在函数调用时进行参数赋值和上下文切换可能影响程序需要的内存大小及其性能。这些都是重要问题。特别是当函数很小，并且需要用大量的局部变量调用时，为了执行数行代码而保存调用方的运行环境，对时间和栈空间浪费的确是太可惜了。

例如考查下面用一个常量系数计算税收的函数。

```
double tax(double gross)
{ return gross * 0.05; }
```

当客户代码调用此函数时，客户函数的“上下文”（即其参数和包括局部数组在内的局部变量）将会保存在栈中。函数调用结束后，再恢复上下文。

```
double sales, state;    ...
state = tax(sales);                  // function call
```

如果可以避免调用如此小规模的函数的额外开销就好了。C语言提供的解决问题的方法是，

为文字上替换文字代码和模拟函数调用使用宏(macro)。

```
#define  tax(x)  x * 0.05
```

有了宏，客户的代码state = tax(sales);将会被宏扩展为state = sales*0.05;这样就避免了一般函数调用的开销。

C++和C一样支持宏功能。但宏是通过预处理程序扩展而不是被编译程序翻译的。宏不是函数。它没有局部变量，不提供参数类型检查，对调试程序也是不可见的。

将宏跨越多行代码进行编写是很糟糕的。当扩展的代码包含语法错误时，编译程序提供的行号是调用宏后的源代码行号，而不是定义宏时的行号。如果宏包括数行，就很难辨别出到底是哪一行导致了错误信息。

宏并不知道C++运算符的级别，它只是做纯粹的字符文本替换工作，而不考虑代码的真正意图。我们来看下面的客户代码。

```
state = tax(sales+20.0);          // expression as the actual argument
```

对程序员来说，这段代码意味着：

```
state = (sales + 20.0) * 0.05;   // desired interpretation
```

但是，预处理器实际上会将这个宏代码按字符替换成：

```
state = sales + 20.0 * 0.05;     // preprocessor interpretation
```

当然，对此也有解决办法（例如在宏定义中使用括号），但这个例子体现出宏替换有一些应该尽量避免的陷阱。在C++中允许我们把函数定义成内联的可扩展的函数。它的功能与宏替换相似，但又克服了#define预处理程序语句的不足。

如果一个函数使用了inline修饰符，则对该函数的任何调用都将被替换成该函数中定义的各个语句，没有函数调用的开销，也不使用栈空间。

```
inline double tax(double gross)
{ return gross * 0.05; }
```

同时，一个内联函数的确是一个函数。它可以有多行代码，可以定义内嵌的语句块，而且可以有其局部变量。作为一个C++函数，内联函数也可以进行参数类型检查和调试操作。

这个功能提供了模块化的优点，而没有（在函数调用开始和结束时的）上下文切换的额外开销。在每个函数调用处把函数体直接插入到客户代码中。这样，有多少处函数调用就在被编译的目标代码中有多少个inline函数的副本代码。

内联函数改善了程序的性能，但如果声明为inline的函数的执行开销不是整个程序执行时间的主要部分，这种性能上的改善可能不明显。同时，内联函数增加了可执行程序的大小，因此可能导致额外的交换从而实际上降低了执行速度。

而且，inline修饰符不是一个无条件指令告诉编译程序要将其代码嵌入客户代码中，它只是提出一个建议。如果编译程序的设计人员认为inline所修饰的函数很长很复杂，编译程序可以忽略这种建议，把它当成一般的函数处理。

有些C++的编译程序不接受有控制结构的内联函数。有些编译程序接受一两个if语句，但不接受循环结构。因此应该只对简单的函数使用内联功能。

对许多函数而言，使用inline修饰符并不能改善程序的性能。应该只对那些其函数调用确实影响程序性能的函数使用inline修饰符。应该通过解决程序的“瓶颈”问题来改善

程序。

在第2章中，我们曾说过定义一个类（或者一个结构）的成员函数有两种办法。一种是在类规则说明的区域内实现函数；另一种是只在类说明的区域内指定函数原型，而在其他地方实现函数。在类的说明中定义的成员函数缺省为inline方式，我们不必再给出inline修饰符。

```
struct Counter {
private:
     int cnt;
  public:
     void InitCnt(int Value)
     { cnt = Value; }              // inline by default
     void UpCnt()
     { cnt++; }
     void DnCnt()
     { cnt--; }
     int  GetCnt()
     { return cnt; } } ;
```

但我们通常只是在类的说明中指定成员函数的原型，而不是函数的实现。

```
struct Counter {
private:
     int cnt;
  public:
     void InitCnt(int);      // prototypes only
     void UpCnt();           // no indication how it is implemented
     void DnCnt();
     int GetCnt(); } ;
```

如果一个类的成员函数的实现代码是在类定义的花括号之外，它就不再缺省为inline方式了。但是可以使用inline关键字定义它为内联函数。

```
void Counter::InitCnt(int Value)
  { cnt = Value; }
inline void Counter::UpCnt()
  { cnt++; }
inline void Counter::DnCnt()
  { cnt--; }
inline int Counter::GetCnt()
  { return cnt; }
```

下一章我们将会讨论更多关于类的知识。

7.5 有缺省值的参数

有缺省值的参数是一种新的语言功能（在C中没有），它的目的是进一步提高程序的可读性和可修改性。声明一个函数时，我们可以为参数表中的一个到多个参数指定缺省值。

这里有一个函数sum()的声明，它计算一个double值数组的各元素之和，数组的元素个数是给定的。函数声明为第二个形式参数使用了初始化语法，以指定其缺省值为25。

```
double sum (const double a[], int n=25);          // a prototype
```

这个初始化语法指示编译程序，如果客户代码中的函数调用没有指定实际值，就使用原

型中定义的缺省值。

```
double total;  double x[100];  int n;  ...        // whatever
total = sum(x);                // add up 25 components of array x[]
```

当然，客户代码也可以显式地提供实际参数的值，以便覆盖缺省值。

```
total = sum(x,n);              // add up n components of array
```

乍一看，这样编写代码似乎没有什么意义。如果希望计算25个元素的和，为什么不能显式地指出呢？而且，使用缺省的参数值会导致一些微妙的细节问题，从而增加代码的复杂性。但在有些情况下使用缺省的参数值可以简化客户代码。例如当函数有大量的参数，而大多数被调用时都是使用相同的参数值，只是偶尔使用别的参数值。例如，iostream.h的函数getline()有如下的原型：

```
istream& getline(char buf[], int count, char delimiter = '\n' );
```

大多数情况下，我们调用此函数只使用两个参数：读入数据的字符数组，当输入流中没有碰到新行分界符之前能够存储的最大字符个数（包括0终止符）。该函数也允许程序员使用任意的分界符，如美元符号、英镑符号或者是其他合适的符号。当以新行为输入分界符而调用函数时，使用缺省值可以能帮助程序员从一次又一次的输入标准的“\n”分界符解放出来。

注意，有缺省值的参数的缺省值应该在函数原型中而不是函数定义中指定。函数定义不应包含缺省参数值。

```
double sum(const double a[],int n)   // no default value is not used
  { double total = 0.0;
    for (int i=0;  i<n;  i++)
      total += a[i];
    return total; }
```

这意味着函数的设计人员可能根本不知道客户代码使用了缺省值。在不同文件中实现的不同函数在声明其原型时，可能为相同的参数指定了不同的缺省值，而没有互相协调一致。

在同一个文件中，一个参数只能使用一个缺省值。当函数在同一个文件中被定义和被使用时，如果在该文件中没有用到函数的原型，则可以在函数定义中指定缺省值。但如果函数原型和函数定义都放在函数被调用的文件中，两者中只能有一个指定缺省值。如果相同函数的两个原型都放在同一个文件中，只能有一个原型指定缺省值。即使两个原型指定的是相同的缺省值，也是语法错误。编译程序并不会比较缺省值是否一样，而只是认为程序员重复定义了缺省参数。

既然函数原型中的参数名是可有可无的，因此将类型名而不是参数名“初始化”为缺省值也是完全可以的。

```
double sum (const double a[], int=25);    // assign to int?
```

我们是在把25赋给int吗？当然不是，这并不是一个赋值语句。只不过是一个标识而已，其目的是告诉编译程序（和维护人员）缺省值的存在。

这是一种典型的C++设计思想。C++极大地扩展了C，因而需要许多新的关键字和运算符。但运算符的数目是有限的，而且在C语言已经用了不少有两个字符的运算符。C++增加了更多的双符号运算符，但是合理的符号组合数目并不是很大。C++还增加一些关键字，但是需要学习更多关键字会使C++看起来很庞大。虽然C++是C语言的超集，但它还是试图成为易于学

习和使用的精简的语言。正因为此，C++只是增加了为数不多的几个关键字（如new,delete,class,public,private和protected等）。也正因为这个原因，在有需要时，C++允许为其他目的重用运算符和关键字。我们已经知道取地址运算符&被重用为引用运算符。在我们的sum()函数原型中，C++也重用了赋值运算符，并给予新的含义：指定缺省的参数值。

这种设计策略是一个不错的主意，它减少了我们需要学习和掌握的符号和关键字的数量，但从另一方面来看，对每个重用的运算符，我们都需要学习它的不同使用，这可能导致混乱。运算符&的重用就是这样一个例子。它的确让程序员感到混乱，特别是那些没有多少经验的程序员。

C++只允许对最右边的参数设置缺省值，而不允许在参数表中间使用缺省值。

```
int foo(int a=0,int b=2,double d1,double d2=1.2);        // no
```

要想通过编译，必须删除最左边的缺省参数值（两个int参数），或者给第一个double参数加上一个缺省值。

这并不是一个很严格的限制，毕竟缺省值总是可以被显式地覆盖。

如果作为缺省值运算符重用的赋值运算符和普通的赋值运算符产生混淆，就会出现问题。例如，我们有如下的一个函数，它动态地创建一个新的结点node（node的类型为Node），并初始化它的信息域（类型为Item）和指向链表结构中下一个结点的链接（类型为Node*）。

```
Node* createNode(Item item, Node* next)
{ Node *p = new Node;                    // allocate heap memory
  p->item = item;  p->next = next;       // initialize node fields
  return p; }                            // pointer for client use
```

在许多程序中，新创建的结点通常是接在一个链表的结尾的，它的下一个域被设置为NULL以表示它是链表的最后一个结点。所以，客户代码在调用createNode()函数时，应该把0（或者NULL）作为第二个实际参数值。

```
tail->next=createNode(item,0);       // append node to list end
tail = tail->next;                   // point to new last node
```

在每次使用createNode()函数作为服务器时，都让客户方负责指定0参数值，这种做法可能不太好。这种责任应该推向服务器，客户代码应该如下所示：

```
tail->next=createNode(item);         // append node to list end
tail = tail->next;                   // point to new last node
```

为第二个参数使用缺省值是一个可行的解决办法。

```
Node* createNode(Item item, Node* next=0);        // prototype
```

然而，如果在函数原型中省略了参数名可能产生意想不到的新问题：

```
Node* createNode(Item, Node*=0);    // what does this mean?
```

这将会产生一个编译错误，编译程序会以为程序员在这里使用了运算符*=，但实际上并不是这样的。让编译程序接受代码的惟一方法是在星号*和等号之间加上一个空格。

```
Node* createNode(Item, Node* =0);  // this is better
```

我们曾说过，C++和C相类似，都会忽略空格吗？是的，C对空格是完全忽略的，但是C++只是承诺忽略空格，仍有例外情况。这些例外是出于不同的目的使用相同的运算符所造

成的。

在应用程序中，如果相同的函数经常使用描述其上下文的相同值的变量调用，缺省的参数值就是很有用的。如果指定的参数值只是在特别的情况下才使用，是否需要使用缺省的参数值就值得讨论。这个问题在进行Windows编程时十分典型。

不加考虑地使用缺省参数值会导致客户代码难于理解，应该尽量避免这种情况。

有时候，使用缺省参数值可能有助于进行代码升级，只需要添加新的代码而不需要修改现有的代码。

让我们来考虑一个简单的函数registerEvent ()，它用于一个实时控制系统。

```
inline void registerEvent()
{ count++;  span = 20; }     // increment event count, set time span
```

当然，真正的系统函数的实现代码要比上面的复杂得多，在这里去掉了无关紧要的细节，只是保留使用全局变量进行事件计数操作和时间片分配的代码。让我们假设它是一个庞大和复杂的系统，要写400页的代码反复调用这个函数。

```
registerEvent();              // server call in client code
```

在系统优化和维护时，不可避免的事情发生了。系统可能需要处理其他类型的事件，需要为这些事件分别设定时间片。原有的400页代码无需修改，因为主要的事件仍然按以前的方式处理，但是我们必须再写10页代码处理主要的事件和新事件。

解决这个问题的一个方法是，写另外一个函数regEvent()。

```
inline void regEvent(int duration)      // another server function
{ count++;   span = duration; }         // increment event count
```

这是一个可行的解决方法，但是仍有不足之处。首先，混合使用registerEvent()和regEvent()函数可能会让人多少有些糊涂。其次，我们需要其他的函数名，这在维护过程中是常见的事情。第三，对类似的操作使用相同的函数名会更好一些。如果我们想要画一个图形或者设置绘画的上下文，对所有形状的绘画函数名应该是draw()和setContext()，而不是诸如draw1()和setContext1()这样糟糕的名字。

所以，照此道理我们应该直接修改registerEvent()函数，给它额外的参数，改变函数体以适应新的需求。

```
inline void registerEvent(int duration)    // we change the header
{ count++;  span = duration; }             // we butcher the body, too
```

现在，我们编写的10页新代码中，使用不同的实际参数值调用registerEvent()函数。

```
registerEvent(50); registerEvent(20);      // new client code
```

与此同时，在原有的400页代码中，对registerEvent()函数的调用也要进行修改。

```
registerEvent(20);            // modified server call in client code
```

所以，这种解决方法需要：

1) 添加新的客户代码。

2) 改变原有的服务器的函数头。

3) 改变原有的服务器的函数体。

4) 改变原有的客户代码。

因为需要在4个地方进行代码协调，我们极有可能把整个系统都弄糟，特别是进行上述的

最后一个操作时。使用缺省的参数值可以提供另一个可行的方法。我们也要修改现有的服务器函数，修改函数头和函数体。

```
inline void registerEvent(int duration)    // we change the header
{ count++;  span = duration; }             // we butcher the body, too
```

但是新的客户代码中和现有的客户代码中的函数原型应该如下所示：

```
inline void registerEvent(int duration=20);        // prototype
```

这就消除了上述列表中大部分繁琐的工作，即需要为了修改其他地方（本例中是修改服务器函数）而修改现有的代码。而这项工作可能正是维护过程中最麻烦的环节。问题并不在于修改时需要大量的劳动力，而是在于必须保证所有需要修改的地方都进行了修改（而且没有对不需要修改的地方进行修改）。这样，为了证明这些改动都是正确的而进行的回归测试是很难计划的，实现也困难，而且几乎不可能文档化。

当然，并不是所有的维护都可以使用缺省的参数值来简化。但是，当可以使用它时，一定不要错过使用它的机会，它显著提高了传统的维护技术。

在下一部分，我们将会学习另外一个C++程序设计技术——函数名重载，它可以用来代替缺省参数值的使用。

7.6 函数名重载

在C++中，函数名重载也是对程序模块化的重大改进。

在大部分的语言中，每一个名字在某一个作用域里面（如语句块、函数、类、文件、整个程序中）都只联系惟一的一个对象。类型名、变量名或者函数名都是如此。

C语言中，嵌套在其他函数中的函数没有嵌套的作用域，它们使用的命名必须在整个程序作用域而不是所在文件作用域中惟一。同一个源文件里定义两个名字相同的函数是语法错误。在两个不同文件中有两个相同名字的函数定义则是连接错误。C语言并不考虑参数类型或者返回值类型。只有函数名才起作用，因此函数名必须在一个工程（包括库）中保持惟一。

在C++中，每一个类都有一个独立的作用域，因此同一个名字既可以用作类的成员函数，也可以用于全局函数。而且，相同的函数名可以用于不同类的成员函数。注意，在不同作用域中有相同函数名的函数并不要求使用不同数目和类型的参数。它们可以相同也可以不同，完全没有关系。一旦两个函数定义在不同的两个作用域中（全局域和类作用域或者两个类作用域），就不会存在名字冲突问题。

C++的这个改进的确是软件开发技术的巨大进步。C语言要求所有函数名都必须是惟一的，这种限制过于严格，特别是对大型工程而言。函数名的大量增加让工程管理变得很困难。对大型工程来说，负责程序的不同部分的工作小组之间很难进行协调统一。在C++中引入类的作用域之后，大部分问题就迎刃而解了。这只是对大多数问题而言，并不是所有问题都解决了。

C++的作用域规则和C的一样：在作用域内，程序员定义的标识符必须是惟一的（类作用域或者文件作用域对应类型名或变量名，工程作用域对应函数名）。如果可以在相同的作用域而不是在不同的作用域中对不同的函数使用相同的名字，那就太方便了。

C++在这个问题上也有一个显著的改进，它允许函数名重载。C++中函数名的意义依赖于函数的参数个数和这些参数的类型。对具有不同个数或者类型的参数的不同函数使用相同的

函数名，这就被称为函数名重载。编译程序能够分辨这些重载函数。

下面的例子为两个不同的函数使用了相同的函数名add()。函数的参数个数不同：一个函数有两个参数，另一个函数有三个参数。

```
int add(int x, int y)                    // two parameters
{ return x + y; }

int add(int x, int y, int z)             // three parameters
{ return x + y + z; }
```

如果几个函数的参数表不同，即使它们有相同的程序员定义的函数名，C++编译程序也会视之为不同的函数。当客户代码调用函数时，客户传递给函数的参数表可以帮助编译程序选择合适的函数定义。

```
int a = 2, b = 3, c, d;    ...       // whatever
c = add(a,b);          // call to:  int add(int x, int y);
d = add(a,b,c);        // call to:  int add(int x, int y, int z);
```

如果参数的个数一样，但它们的类型不同，则参数表也不相同，因此函数名重载也仍然可行。

```
void add(int *x, int y)                  // also two parameters
{ *x += y; }
```

add()函数也有两个参数，但是第一个参数有不同的类型：它是一个指向整数的指针，而不是整数类型。这就足以让编译程序分辨出不同的函数，因为客户代码中的函数调用是不同的。

```
int a = 2, b = 3, c, d;    ...       // whatever
c = add(a,b);          // call to:  int add(int x, int y);
d = add(a,b,c);        // call to:  int add(int x, int y, int z);
add(&a,b);             // call to:  void add(int *x, int y);
```

从这里我们可以知道，C++的函数调用的意义是由其上下文决定的：即由客户代码提供的实际参数类型决定。为了解决二义性问题，C++编译程序使用函数标识（function signature）。函数标识的另一个术语是函数的公共接口。它基于函数参数的个数及其类型。参数类型的顺序不同就足以表示函数不同。

当然，参数名在分辨重载函数时是不起作用的。

在区分重载函数时，return的值也是不列入考虑之列的。函数的不同必须表现在参数类型或者参数个数上。

```
double add(int x, int y)    // signature is the same: syntax error
{ double a = (double)x, b = (double)y;
  return a + b; }           // return type is different: not enough
```

C++编译程序并不能区分这个函数和第一个返回是int的add()函数。

```
int a = 2, b = 3, c, d;  double e;    ...        // whatever
c = add(a,b);                // call ambiguity: which function?
e = add(a,b);                // call ambiguity: which function?
```

对于我们来说，第一个调用用来设置整型变量的值，第二个调用设置双精度类型变量的值，这足够区分两个函数了。但是对于编译程序来说，它无法区分这种不同。

如果这两个函数add()定义在不同的文件中，但都在相同的客户文件中进行调用，编译

程序将会标志第二个原型为语法错误，因为编译程序认为它试图重新定义已经定义的函数。

```
int add(int x, int y);       // a legitimate prototype
double add(int x, int y);    // function redefinition: syntax error
```

注意，如果返回值类型匹配，编译程序将接受第二个原型，认为它是对函数的简单重复声明。

```
int add(int x, int y);       // a legitimate prototype
int add(int x, int y);       // function redeclaration: no problem
```

想通过函数的实现代码的不同来区分不同的函数也是不行的，使重载的函数具有语义上相似的操作（这正是它们相同的名字所暗示的）是程序员的职责。例如，我们可以编写另一个add()函数，它有4个形式参数，返回实际参数的最大值。

```
int add(int a, int b, int c, int d)     // yet another overloaded add()
{ int x = a>b?a:b, y = c>d?c:d;         // bad use of conditional operator
  return x>y ? x : y; }                 // return maximum value
```

对于编译程序来说，上面的函数是完全合法的，也可以通过它们的接口与其他版本的add()函数相区别。但对于主管（或者维护人员）来说，看到这样的代码，他们会怎么想？

使用重载函数，就不需要为不同但是相关的函数设计惟一的函数名。

```
int addPair (int, int);          // instead of int add(int x, int y);
int addThree (int,int,int);      // instead of int add(int,int,int);
void addTo (int *, int);         // instead of void add(int *, int);
```

这样修改后，无论是编译程序还是维护人员都可以很容易地区别这几个函数。

如果编译程序不能将实际参数和给定函数名的任何形式参数集合相匹配，就会出现语法错误。如果不能精确地匹配参数类型，编译程序会使用类型提升和隐式类型转换。在下面的例子中，假设Item是一个与int类型不相容的结构类型。

```
int c;  Item x;     ...          // whatever
c = add(5,x);                    // no match: syntax error
c = add(5,'a');                  // no error: promotion
c = add(5,20.0);                 // no error: conversion
```

在需要整数进行计算的地方使用字母代替并没有多大意义。这种用法应该是非法的，但实际上是合法的，因此尽量避免使用这种类型提升或者转换，除非确实有其益处。（坦白说，本书并不认为这样做有什么好处，但是为了避免武断而不使用“没有任何好处”这样的字眼。）

注意 对于类参数来说，如果定义了转换运算符和/或者转换构造函数，C++编译程序也可以应用程序员定义的类型转换。（后面将详细讨论）

当两个重载函数有相同个数的参数，而参数类型之间允许互相进行类型转换时，为了避免类型转换的二义性，最好提供类型精确匹配的实际参数。让我们来看两个max()函数，一个有整数类型参数，另一个有double类型参数。

```
long max(long x, long y)              // return the maximum value
{ return x>y ? x : y; }

double max(double x, double y)        // it is different from long
{ return x>y ? x : y; }
```

如果实际参数的类型和形式参数的类型精确匹配，C++编译程序不难决定客户代码的函数

调用使用哪个正确版本的函数。

```
long a=2, b=3, c;
double x=2.0, y=3.0, z;
c = max(a,b);       // no call ambiguity: long max(long, long);
z = max(x,y);       // no call ambiguity: double max(double, double);
z = max(a,y);       // ambiguity: which function?
```

在最后一个函数调用中，第一个实际参数的类型是long，第二个实际参数的类型是double。虽然其返回值类型是double，但编译程序仍无法判断应该调用哪个函数。我们可以显式地将实际参数强制转换为合适的类型来通知编译程序。

```
z = max((double)a,y);  // no ambiguity: double max(double,double);
```

在下一个例子中，我们尝试传递int类型的变量。显然，从int类型转换为long要比从int类型转换为double类型更自然些，不是吗？不对，这对我们人来说可能是自然的，对C++的编译程序来说就不是了。C++并没有类型“亲密”性的概念。类型转换就只是类型转换而已。

```
int k=2, m=3, n;
n = max(k,m);           // ambiguity: which function? long? double?
```

在编译程序看来，把int转换为long或者把int转换为double是完全一样的。既然是一样的，编译程序就会将这个调用标记为二义性。

由此可见，使用函数重载这样好的特性会导致明显的程序复杂性问题。或许使用maxLong()和maxDouble()两个函数并不是一个坏主意。特别是，复杂性问题还没讨论完。让我们再来看另外两个重载函数。

```
int min (int x, int y)              // return the minimum value
{ return x>y ? x : y; }

double min(double x, double y)      // it is different from int
{ return x>y ? x : y; }
```

这样做编译程序又不知所措了。我们知道应该如何做，但是在编译程序看来，从long转换为int或者从long转换为double是一样的。因此函数调用又是语法错误。

```
long k=2, m=3, n;
n = min(k,m);         // ambiguity: which function? int? double?
```

如果我们使用short和float类型的实际参数会怎么样呢？可能有人会说编译程序的编译结果一样，又是二义性的函数调用。情况并非如此，编译程序顺利地编译下面这段代码。

```
short a=2, b=3, c;
float x=2.0f, y=3.0f, z;
c = min(a,b);         // no call ambiguity: int max(int, int);
z = min(x,y);         // no call ambiguity: double max(double, double);
```

这里之所以没有错误，是由于没有进行类型转换。short类型的值被提升而不是被转换为int类型；类似地，float类型的值被提升而不是被转换为double类型。经过提升，编译程序就可以精确地匹配实际参数类型和形式参数类型。所以在这里没有二义性。

当参数按值传递时，const修饰符被认为是多余的或者是无关紧要的。因此它不能用来区分重载函数。例如下面的函数就不能与前一个函数int min(int ,int)区分开。

```
int min (const int x, const int y)       // return the minimum value
```

```
{ return x>y ? x : y; }
```

类似地，从一个类型到一个引用的转换也是无关紧要的，也不能用来区分重载函数，因为这些函数的函数调用形式看起来是一样的。例如，编译程序不能分辨下面这个函数和函数int min (int , int)。

```
int min (int &x, int &y)                    // return the minimum value
{ return x>y ? x : y; }
```

另一方面，编译程序很容易区分指针和被指向的类型，如int*和int。而且编译程序也可以区分常量和非常量指针和引用。为了进行演示，我们看看下面两个小函数：

```
void printChar (char ch)              // value parameter
{ cout << ch; }

void printChar (char* ch)             // pointer parameter
{ cout << *ch; }
```

客户代码中的函数调用看起来是不同的。当前面两个函数调用使用了一个普通的字母（非常量）时，编译程序和程序员都可以区分它们。

```
char c = 'A';  const char cc = 'A';
printChar(c);        // ok: void printChar(char);
printChar(&c);       // ok: void printChar(char*);
printChar (cc);      // const can be passed to void printChar(char);
printChar(&cc);      // const cannot be used in printChar(char*);
```

第3个调用也是可接受的，因为在函数期望一个非常量的值参数时，可以传递一个常量参数。如果函数修改了参数的值，这种修改也不会扩展到客户内存空间，也就不会修改常量参数的值。第4个调用有语法错误。如果函数修改了它的（按指针传递过来的）参数，这种修改将扩展到客户内存空间，因为实际参数被声明为常量，因此不能在这个函数调用中使用。

是不是认为这些微小而又烦人的规则难以理解？让我们再为这个函数集合编写一个重载函数。在这个函数中，函数头反映了函数体的功能，函数不会修改实际参数。

```
void printChar (const char* ch)       // pointer, but value is const
{ cout << *ch; }
```

现在上面的4个函数调用都可以通过编译并且正确执行了。注意如果没有第2个函数（void printChar(char*);），第2个函数调用仍然通过编译，它会调用void printChar(const char*);在需要const值的地方传递一个非常量值也是合适（和安全）的。

也要注意，在双引号中的具体字符串是char*类型，而不是const char*，因此我们可以设置普通的指针指向它们，然后通过这些指针对它们进行修改。

```
char *p = "day"; p[0] = 'p';        // now it says "pay"
```

相同类中的成员函数重载和相同文件中的函数重载有相同的规则：如果参数个数不同或者参数的类型不同，为不同函数使用相同的函数名也是合法的。如果在相同的类中重载了成员函数，它们应该有相似的语义。（当然，编译程序不会对此进行检查。）类的构造函数常常被重载。这种重载让客户可以在不同的上下文中有选择性地初始化对象。（我们将会在下一章讨论具体内容）。

虽然我们用来演示函数名重载的例子很简单，但是足以说明使用重载可能让客户代码难

以理解。决定该调用哪个函数，对编译程序来说都是困难的，更会让阅读代码的人感到糊涂。C++中常常出现这种复杂的情况。应该尽量少用这种特性

函数重载也可以和函数缺省参数一样，用于软件改进。当程序功能改进时，我们修改现有的函数以满足新的需求。这种方法通常要求在函数接口、函数体和调用该函数的客户代码中进行修改。因此这种方法很复杂，易于出错，开销很大。

在某些场合，函数名重载允许我们只用添加新的服务器函数，而不用修改现有的服务器函数和客户函数调用。我们再来回顾简单的函数registerEvent()，我们曾用它来演示缺省参数值的使用。

```
inline void registerEvent()
{ count++;  span = 20; }     // increment event count, set time span
```

再次假设它是一个庞大复杂的系统，包含400页之多的代码会调用到上面的函数。

```
registerEvent();             // server call in client code
```

现在想要编写10页新代码，为不同的事件分别设置时间片。原有的400页代码无需改动，因为时间片是保持不变的。

当然，编写其他函数例如regEvent()来为这10页代码服务，也是可行的选择。

```
inline void regEvent(int duration)      // another server function
{ count++;   span = duration; }         // increment event count
```

同样地，这是个小例子，从头开始编写这个小函数毫不困难。但在实际生活中，函数是长而复杂的，使现有的函数适应新的情况的想法总是很强烈的。让我们修改现有的registerEvent()函数，增加它的参数个数并相应地修改它的函数体。

```
inline void registerEvent(int duration)    // we change the header
{ count++;  span = duration; }             // we butcher the body, too
```

我们在前面也说过，这种方法需要：

1) 添加新的客户代码（如我们假想的10页代码）。

2) 改变原有的服务器函数（添加新的参数）。

3) 改变原有的服务器函数体（使用新的参数）。

4) 修改已有的客户代码（如我们假想的400页代码）。

使用缺省参数值，我们就无需修改现有的客户代码，但是还要修改现有的服务器函数和它的接口。使用函数重载，我们就无需修改现有的registerEvent()函数。（它和最后那个函数看起来完全一样。）

```
inline void registerEvent(int duration)    // new function header
{ count++;  span = duration; }             // new function body
```

现在，通过函数重载我们只需两方面的改动：

1) 添加新的客户代码（如我们假想的10页代码）。

2) 增加新的服务器函数。

这样，现有的客户代码和服务器函数都不需要进行修改了。非常好！但是，并不是每个维护任务都可以使用这个方法。如果可以一定不要放弃使用这个方法的机会。它显著提高了传统的维护技术。

7.7 小结

在本章中，我们将C++函数看做构造程序的主要工具。C++是从C语言发展而来的，在众多的现代高级语言中，C++是很独特的。因为它要求程序员为每个源代码中使用的函数提供原型。这个规则支持单独编译，从而支持大型工程项目的管理，但是它也给设计人员和维护人员创建了额外的程序。

C++中的函数参数传递是复杂的技术。软件开发人员必须在4个地方协调统一代码：在客户代码中（在函数调用本身）、在函数头中、在函数原型和服务器函数体中。常常是一个地方出错导致各种更加严重的问题。

按值传递参数确实比较简单，但它不能修改实际参数的值。按指针传递支持客户代码的副作用，但是很复杂而且容易造成错误。C++从C中继承这两种参数传递方式。为了减少出错的机率，C++试图减少按指针传递参数的使用。于是又引入了另一种参数传递方式：按引用传递。这看起来是很好的折中，虽然这种参数方式在引入一些术语上和标识上有混淆。

对于结构类型来说，按值传递有另外一个不足：将实际参数复制到为函数参数分配的栈空间中，需要额外的时间和空间。按引用传递可以避免复制，同时又没有按指针传递的额外复杂性。但是按引用传递又很难将代码设计人员的意图传达给维护人员，即函数修改了哪些参数，没有修改哪些参数。使用const修饰符可以解决这个问题。这是非常有用的方法。

对于数组类型来说，只有一种参数传递模式，而且输入和输出参数的语法形式是一样的。这又会让维护人员很难理解程序中的数据流：函数修改了哪些参数，哪些参数保持原来的值。

使用const修饰符允许代码的设计人员告诉维护人员哪些数组没有作为函数调用的结果被修改。但是，不分青红皂白地认定没有const修饰符的数组就会被函数修改也并不总是安全的，设计人员应该确保让维护人员的确从这种方法中受益。

我们还讨论了参数提升和转换。当实际参数类型和形式参数类型不相容时，既不允许提升也不允许转换。如果类型属于不同的类别就是不相容的，这些类别包括标量值、指针、结构和数组。它们之间是不能转换的。不同类型的结构之间同样不能转换。在这些情况下，C++是强类型语言。但是，C++允许在标量数值之间进行隐式的类型转换，这没有任何问题。此外，C++也允许在不同类型的指针或者不同类型的数组之间进行显式类型转换（或强制类型转换）。这些转换为程序员提供了很大的灵活性，但是容易出错而且可能使维护人员觉得困惑。

我们还学习了内联函数，它消除了函数调用的性能开销。正确地使用它可以改善程序的性能；不正确地使用它可能增加目标代码的大小，甚至因为额外的交换而破坏程序的性能。

除此之外，我们还讨论了函数的缺省参数值和函数名重载。这些技术都是很好的语言特性，极大地缓解了对C++程序设计工程中的名字空间的压力。它们甚至为程序的维护开辟了新的领域，当被调用的函数需要修改时，不再需要修改现有的客户代码。但是，这些特性应该尽量少使用，因为它们十分复杂，还有许多不为人所注意的特殊情况。任意地使用这些特性很容易让编译程序和维护人员感到迷惑。

用函数编程的技术是C++程序设计的支柱。如果不能熟练地使用C++函数，就不可能创建出高质量的面向对象程序。在下一章，我们将开始学习面向对象的程序设计技术——创建高质量程序的最强有力的工具。

第8章 使用函数的面向对象程序设计

这一章，我们开始学习面向对象程序设计的原则和技术。所涉及的内容有些是通用的程序设计知识，有些是我们专门为C++的使用而阐述的。这些原则和技术在其他C++的书中很少讨论，因此，即使你是一个经验丰富的C++程序员，我们也建议不要跳过这一章。

前几章我们着重于讨论C++语言规则，它定义了在C++程序中什么是语法上合法的和不合法的。与自然语言相似，我们应该排除不合法的用法，这并不是因为风格不好或者产生二义性，而是因为编译程序无法将不合法的代码转换成目标代码。即使是合法的用法，也有很多不同的方法“表示相同的事物”。前几章中，常常从程序正确性、性能、风格好坏等角度对合法的不同方法进行比较。但是我们主要关心的应该是程序的可维护性，即确保维护人员不会花费额外的精力去理解代码设计人员在编写源代码时的思路。

在这一章（和下一章），代码的可理解性将成为我们关心的主要因素。然而，讨论的重点会从编写某一个代码段的控制结构转到更高层次的程序设计，即把一个程序分割成协作的多个部分（如函数或者类）。

我们不会进行系统分析，即决定在程序中应该使用哪些函数来支持应用程序的目标。这样会使本书的覆盖面太广。因此，我们假设为了实现程序目标所必需的所有函数已经存在了。这样，我们就可以集中注意力考虑：应该采取什么样的方法，使用额外的函数让程序的可维护和可重用性更好。

在互相合作以实现程序目标的客户函数之间分配工作的方法一直都不止一种。设计处理数据和代表客户函数执行操作的服务器函数的方法也不止一种。如果所有的方法从程序正确性角度来看都是等效的，我们应该如何判断哪个方法更好呢？

过去，大多数程序员会使用程序性能作为主要标准。但是硬件上的进步让这个标准不再适用于许多应用程序，特别是那些交互式应用程序。对那些性能仍然很重要的应用程序来说，影响性能的是算法和数据结构，而不是在客户函数和服务器函数之间分配工作的方式。

另一个重要标准是代码的易编写性，这对于那些由少数几个人开发的小型程序来说还很适用，这些代码通常只使用很短的时间，过了一段时间就会被丢弃或者被全新的代码代替。对于由很多协作的开发人员设计并在很长时间内都进行维护的大型系统而言，考虑到软件开发的经济性，这个评价标准就不适用了。程序最好的版本是其组成部分容易重用（在系统开发或将来的发布过程中节省开支）或易于维护（在程序的改进过程中节省开支）的版本。

这两个标准——可维护性和可重用性，是评估软件质量的最重要依据。当然，这两个依据很空泛，实际上也没有很明确的方法判断哪个版本的代码维护或重用起来比较便宜。

可重用性与程序各个部分之间的独立性有关。在C++不同版本的代码中，和程序的其他代码段连接较少的版本在其他上下文中更加容易重用。可维护性同样与程序各个部分之间的独立性有关。在C++代码的不同版本中，如果一个版本花较少时间就可以理解，而且还不需要研究程序的其他代码段的版本，这样的版本就更加容易修改，同时不会对代码的其他部分产生副作用。

这就是为什么说需要参考程序的其他代码段才能理解的代码其质量差的原因。这也是为

什么说只需要单独理解一段代码而不需要参考程序的其他代码就可以证明代码质量好的理由。如果一个版本的代码可以花费较少的精力去理解，而且可以少参考代码的其他部分，我们就常常说这个版本的代码比另一个版本的代码要好。

理解这些判断标准是件好事，但是对初学的程序员来说还不够具体。代码可理解性和独立性的概念应该由其他更加专门的、更加容易辨识和使用的技术指标来支持。在本章，我们提供了几个技术标准。其中的两个标准是比较旧的内聚性和耦合度，还有两个标准是比较新的数据封装和信息隐藏，即使是业界也没有积累使用这两个新概念的足够经验。除了封装和信息隐藏以外，我们还将使用与代码可理解性和独立性相关的几个标准：

- 将职责从客户函数推向服务器函数。
- 限制服务器函数和客户函数共享的信息。
- 避免将应该属于一起的部分拆分开来。
- 应该在代码中而不是在注释中将开发人员的意图传达给维护人员。

我们尚未找到单一的术语包含这些原则（是使用“最大独立性原则”？“Shtern原则”？“在需要知道的基础上共享知识”？还是“自解释代码原则”？）。正如我们将要看到的那样，这些原则之间相互重迭，也和内聚性、耦合度、数据封装和信息隐藏相交叉。初学者应该熟悉所有这些原则。它们的主要优点是操作性强，可以具体地告诉程序员从什么方向寻求更好的设计。使用这些原则可以帮助我们理解如何提高编码质量。

这些标准和原则背后的思想是，程序中的函数合作完成相同工作的各个部分。无论在它们之间如何划分功能，它们都必须共享信息，有共同的关注点，参与相同工作的各部分。这些函数是相同程序的组成部分。为了增加函数的可重用性和可理解性，我们在函数之间分配任务时（设计系统时）要采取使函数之间依赖性最小的方法。

编写一个较好的程序和进行高质量的程序设计一样，比编写低质量的程序需要更多的时间，生成更多的源代码。因此，一些程序员（和管理者）可能对工作量的增加感到失望。我们可以用交通规则进行类比以说服这些程序员（和管理者）。

看到红灯时我们必须等待。我们有时候想，如果没有交通灯也许能更快地到达目的地。的确，对于某些目的地和司机来说是这样，但并不是对所有的目的地和司机都这样。没有交通规则更容易造成交通事故或者交通阻塞，那些没有经过事故地点的司机可能很快就到达目的地，但其他碰到事故或者阻塞的司机就会被耽误。交通规则强迫我们牺牲暂时的时间来从整体上节省时间。

类似地，无视可维护性和可重用性等规则的存在可能会使一些程序员在开发某些应用程序时能更快地编写代码。但对于所有的应用程序和所有的程序员来说就不是这样了。在编写难以理解的程序时节省的时间，可能不足以弥补为了推测出代码的设计人员在编写代码时的目标（和设计人员出错的地方）而花费的时间。

这就是软件业强调要写注释的原因。我们现在投入精力写注释，从长远的角度来看可以给我们带来很多好处（特别是当注释清晰、完整、更新及时的时候）。通常，代码的行注释很模糊、不完整、不能反映代码编写后进行的修改。因此，花精力编写自解释的代码比写注释更好。

对于一个小型应用程序，编写自解释性的代码的原则还显得不那么重要。如果开发一个大型软件系统，为了获得长远的益处而编写高质量的代码就很关键了。

8.1 内聚性

内聚性描述了设计人员在同一个代码段（例如函数）中放置的操作步骤之间的相关性。

如果一个函数内聚性好（高内聚性），它通常只是对一个计算对象或数据结构执行一个任务；如果函数内聚性差（低内聚性），这个函数可能对一个对象执行几个任务，或者甚至对几个对象执行多个任务。一个函数内聚性很差时，会包含对彼此无关的可计算对象进行的无关计算。这意味着这些对象属于其他地方，但设计人员把它们从应该属于一起的其他东西拆分开来，而没有将属于一起的东西放在同一个函数中。

高内聚性的函数容易命名，这些名字通常是动词+名词结构。动词用来描述该函数执行的行为，名词则说明行为的对象（或者主语）如insertItem()、findAccount()等（前提是函数名能够名副其实，但实际情况并非如此）。

低内聚性的函数名通常会使用几个动词或者名词，如findOrInsertItem()等。

下面这个例子就很糟糕。（内聚性差的典型例子都是很糟糕的，因为它们描述的是设计很差的函数。）

```
void initializeGlobalObjects ()
{ numaccts = 0;                          // one computational object
  fstream inf("trans.dat",ios::in);      // transaction file
  numtrans = 0;                          // another computational object
  if (inf==NULL) exit(1); }              // transaction file again
```

在上面的例子中，numaccts应该在处理账号时初始化——因为它属于账号处理。类似地，numtrans则应该在交易处理时初始化——因为它属于交易处理而不是账号初始化。在这个函数中，我们把属于其他处理步骤的操作拆分开来，堆砌在一起组成一个内聚性弱的函数。

补救方法就是重新设计。我们在第1章曾提到，重新设计意味着改变程序组成部分（函数）列表及其职责。对于一个内聚性差的函数，重新设计意味着把内聚性差的一个函数分割成几个内聚性好的函数，其代价是可能产生过多的小函数。除了对性能造成潜在的影响外，还导致维护人员必须记住大量的函数（函数名及其接口）。像上面的函数initializeGlobalObjects()，即使分成几个部分也没有什么意义。我们应该尽量避免编写类似的函数。

内聚性并不是一个很重要的标准，因此不应该轻易决定通过分割函数来进行再设计。在犹豫的时候，内聚性还需要其他标准进行补充。但是，内聚性仍然是评估软件设计的重要标准。我们在评价不同的设计方案（即如何在函数之间分配工作）时应该使用它作为标准之一。

8.2 耦合度

耦合度是比内聚性更重要和更有用的评估标准，它描述了被调用函数（服务器函数）和调用函数（客户函数）之间的界面或者说数据流。

耦合度可以是隐式的，即通过全局变量进行函数间交流；也可以是显式的，即客户函数和服务器函数通过参数进行交流。隐式耦合度的耦合度高，它导致了客户函数和服务器函数之间较高级别的依赖性。显式耦合度的耦合度较低，当函数通过参数交流时，比较容易理解、重用和修改。

耦合度通过客户函数和服务器函数之间来回传递的值的个数来描述。传递值的个数越多，

说明耦合度越强，函数间依赖程度越高；传递的值的个数少，说明耦合度弱，客户函数和服务器函数之间的依赖程度较低。

8.2.1 隐式耦合度

客户函数为服务器函数提供用来计算的输入数据，并依赖于服务器函数计算的结果（服务器输出）。如果函数间的交流通过没有在服务器函数接口中列举出来的全局变量完成，这种耦合度方式就是隐式耦合度。

考虑一个交互式应用程序，它提示用户输入年份，然后输出判断是不是闰年。

```
int year, remainder;  bool leap;                    // program data
cout << "Enter the year:  ";                        // prompt the user
cin >> year;                                        // accept user input
remainder = year % 4;
if (remainder != 0)                                 // it is not divisible by 4
     leap = false;                                  // hence, it is not a leap year
else
   { if (year%100 == 0 && year%400 != 0)
        leap = false;                               // divisible by 100 but not by 400
     else
        leap = true; }                              // otherwise, it is a leap year
if (leap)
  cout << year << " is a leap year\n";              // print results
else
  cout << year << " is not a leap year\n";
}
```

这个程序和我们在第4章里讨论的代码（程序4-8和程序4-9）相似。这的确是一个很小的程序，不需要任何模块化设计。从模块化设计中得到最多好处的一般都是大型程序。研究程序细节和比较不同的解决方案会成为主要任务，干扰我们讨论模块化原则，而后者才是我们需要集中注意力研究的。我们在实际工作中需要应用的是这些原则而不是范例的细节。

因此我们假设这是一个很大很复杂的程序，经过多个周期的重新设计将它分割成互相合作的函数。

这样就有了一个庞大的程序，我们要将它分割成可管理的组成部分。同样是出于简单起见，我们只分割成两个函数，main()函数负责用户界面操作和一般的数据计算流，isLeap()函数使用year和remainder的值来计算leap的值，leap被main()函数用来输出最终结果。

```
void isLeap()
  { if (remainder != 0)          // it is not divisible by 4
      leap = false;              // hence, it is not a leap year
    else if (year%100==0 && year%400!=0)
      leap = false;              // divisible by 100 but not by 400
    else
      leap = true; }             // otherwise, it is a leap year
```

这里有一个技术问题和我们在第6章讨论的作用域概念有关。在main()中设置函数isLeap()使用的变量是year和remainder，而main()也要使用函数isLeap()计算的leap值。如果我们在main()中定义这些变量，它们就只能在main()中可见，C++作用域规则会阻止其他任何函数访问这些变量，因此isLeap()不能操纵它们。如果在isLeap()

中定义它们，那么只是在isLeap()中可见，C++作用域规则会阻止main()函数访问这些变量。为了让main()和isLeap()都可以访问这些变量，我们将它们定义为全局变量。

程序8-1演示了这个解决方案，其运行结果如图8-1所示。

程序8-1　通过全局变量隐式耦合度的例子

```
#include <iostream>
using namespace std;

int year, remainder;                    // global input variables
bool leap;                              // global output variable

void isLeap()                           // inputs: year, remainder; output: leap
   { if (remainder != 0)                // access three global variables
       leap = false;                    // if not divisible by 4, it is not leap
     else if (year%100==0 && year%400!=0)      // access global variables
       leap = false;                    // divisible by 100 but not by 400: not leap
     else
       leap = true; }                   // otherwise, it is a leap year

int main()
{ cout << "Enter the year:  ";
   cin >> year;                         // prompt the user, enter data
   remainder = year % 4;                // access global variables
   isLeap();                            // define whether it is a leap year
   if (leap)
     cout << year << " is a leap year\n";      // print results
   else
     cout << year << " is not a leap year\n";
   return 0;
   }
```



图8-1　程序8-1的输出结果

在这个程序中，函数main()调用函数isLeap()。main()是客户函数，需要调用其他函数完成工作。函数isLeap()是服务器函数，为调用它的客户完成一些工作。这两个函数之间的关系如图8-2中的对象图所示。对象图还显示了函数间的数据流。变量year和remainder在main()中设置，并作为isLeap()的输入值来计算结果。变量leap的值由函数isLeap()计算出来作为结果，并在main()调用isLeap()后使用

注意，输入变量year和remainder在main()调用的isLeap()函数前就必须具有合法的值。由客户函数保证这些变量被恰当地初始化，因为isLeap()中并没有做任何的有效性检查，它只是假设客户函数履行了它的职责。

类似地，输出变量（本例中是变量leap）在函数调用前没有合法的值。设置输出值是服务器函数的职责，客户函数将在函数调用后（而不是调用前）使用这个值。

理解函数之间的数据流是很重要的。如果我们知道变量year和remainder作为isLeap()的输入变量，就会认为函数只是使用它们的值而不会修改它们。所以函数

isLeap()不应改变这些变量的值。

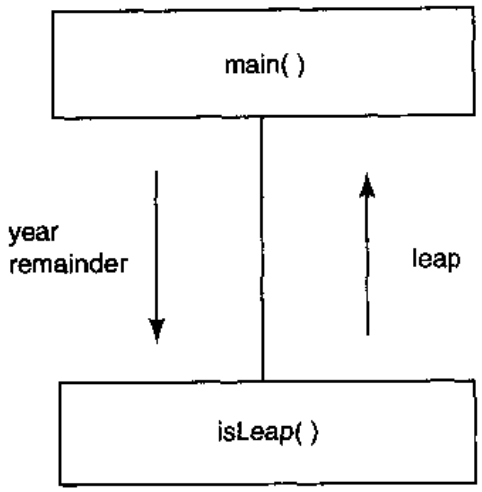


图8-2 程序8-1的对象图

```
void isLeap()
  { remainder = 4; year = 2000;  . . .      // unexpected nonsense!
```

同样，如果我们知道变量leap是函数isLeap()的输出变量，我们也不会期望客户函数main()在调用isLeap()之前初始化这个变量（或者，我们也不期望客户代码在调用isLeap()后不将返回值用作其他用途就立刻改变它的值）。

```
int main()
{ cout << "Enter the year:  ";
  cin >> year;                  // prompt the user, enter data
  remainder = year % 4;         // access global variables
  leap = false;                 // misleading initialization before call
  isLeap();                     // define whether it is a leap year
  leap = true;                  // misleading (and incorrect) if done after call
  . . .
```

当一个维护人员阅读上面的代码时会怎么想？在确定了给remainder赋值的目的后（即用于isLeap()计算变量leap的值），维护人员会再次研究isLeap()函数代码，以弄清楚给leap进行赋值的目的。对一个小函数，只要花几秒钟就可以分析出在客户函数main()中赋给leap的值没有在服务器函数isLeap()中使用，甚至都没有在客户函数main()中使用。但只是对小函数才这么容易弄清楚，对于一个大型程序而言，推断这些信息需要更多的时间，而且维护人员可能感到很迷惑、从而得出错误的结论。

的确，有些程序员十分不喜欢使用未初始化的变量，即使没有必要也会初始化所有的变量。他们认为如果服务器函数由于某种原因没有进行赋值，提前赋值就很有用。但是isLeap()并不属于这一类函数！我们已经编写和即将编写的大多数函数都不属于这种情形。如果程序员理解函数之间的数据流，他们就永远不会编写出忘记对输出变量赋值的函数。

我们看到，这些看起来很无辜的“防备性”的程序设计技巧却使代码需要更多的时间来理解。从评估软件质量标准的角度（可读性和程序各模块间的独立性）来看，这种方法无一例外地产生较差的代码，即它可能会导致一些软件问题，而这些问题是我们希望消除的。为了避免这种情况，我们不应该初始化任何变量，而应该告诉维护人员哪些值是用来作为服务

器的输入变量（在客户函数中初始化它们），哪些值是作为服务器的输出变量（不要在客户中初始化它们）。

希望大家能够领会上述讨论的意义，并且明白将开发人员关于函数间数据流的知识传达给维护人员的重要性。下面我们再回过头来继续讨论耦合度。

耦合度需要我们花费较大的精力去弄懂函数间的数据流。通常我们要研究客户函数和服务器函数采用的数据处理模式。例如在程序8-1中，我们注意到在main()中对变量year和remainder进行了赋值，然后isLeap()使用了这些值。我们也注意到main()没有初始化leap，而isLeap()函数给leap赋了值，然后main()在调用isLeap()后使用了这个值。

然而，为了弄清楚这些简单的依赖性，我们必须完整地研究客户函数和服务器函数。对我们讨论的小例子来说这是很简单的，但是对具有一定现实性大小和复杂性的函数就需要花更多的时间研究。用什么方法可以解决这种劳动密集型且容易出错的问题呢？方法就是使用显式耦合度取代隐式耦合度。

8.2.2 显式耦合度

显式耦合度通过函数的参数来实现，这时服务器函数使用的所有输入和输出变量都包含在服务器函数的参数中，不需要在客户和服务器之间的数据流中使用全局变量。程序8-2所实现的功能和程序8-1完全一样，只是用显式的参数取代通过全局变量进行的隐式数据流。这个程序的执行方式和程序8-1的一样。

程序8-2　通过参数进行显式耦合度的例子

```
#include <iostream>
using namespace std;

void isLeap(int year, int remainder, bool &leap)    // parameters
// inputs: year, remainder; output: leap
  { if (remainder != 0)
      leap = false;
    else if (year%100==0 && year%400!=0)
      leap = false;
    else
      leap = true; }
int main()
{ int year, remainder;                    // local input variables
  bool leap;                              // local output variable
  cout << "Enter the year:  ";
  cin >> year;                            // input variables are set
  remainder = year % 4;
  isLeap(year,remainder,leap);            // output variable is set
  if (leap)                               // output variable is used
    cout << year << " is a leap year\n";
  else
    cout << year << " is not a leap year\n";
  return 0;
  }
```

程序8-2中，函数isLeap()有3个参数，而没有全局变量。变量year、remainder和leap都在客户函数mair()中定义为局部变量。为什么这样呢？因为它们不需要像在程序8-1中那样在函数isLeap()的作用域中也可见。函数isLeap()将它们作为实际参数来访问，

实际参数在函数isLeap()的调用过程中从客户函数传入。

综上所述，当两个函数通过数据进行交流时，数据流的元素只有两种定义方式，或者声明为相对两个函数都是全局的变量，或者在客户函数的作用域中定义，然后作为参数传递到服务器函数。

在上面的例子中，变量year和remainder是isLeap()函数的输入变量，而变量leap是输出变量。我们是怎么知道的呢？我们研究函数isLeap()的头部（或者原型——任意一个都可以）而不是函数体而推理得到。

```
void isLeap(int year, int remainder, bool &leap)        // parameters
{ . . . }
```

我们怎么可以不看函数代码就能知道每个参数的作用呢？当然可以，因为参数year和remainder是按值传递，所以它们不可能作为输出变量，函数体内也不应该给它们赋值。

```
void isLeap(int year, int remainder, bool &leap)    // parameters
{ remainder=4; year=2000; . . .                     // useless for value parameters
```

由此，我们可以得出这两个变量是输入参数的结论。客户代码应该在函数调用前设置实际参数的值，而且服务器函数会用这些值进行计算。

类似地，leap按引用传递，这意味着它是输出参数。实际上它也有可能既是输入参数又是输出参数。即客户函数可能先设置它的值，而服务器函数可能会修改它的值。但重点还是isLeap()函数修改了参数leap的值。

我们要研究多少代码才能得到以上的结论呢？不多，只用看函数头就可以了。程序8-2的结构如图8-2所示，这和程序8-1的是一样的。只是全局变量的显式数据流由参数的显式数据流代替。我们进行代码研究所花费的时间依赖于客户函数的大小或复杂性吗？不是。那么依赖于服务器函数的大小或复杂性吗？也不是。从隐式耦合度转换到显式耦合度，对维护人员和设计人员来说都极大地减低了代码复杂性。

这个例子说明了我们为什么要尽量避免使用全局变量。要不要使用全局变量从业界开始争论这个问题到现在已经有三十多年了，但是很多程序员还不大清楚其问题所在。我们经常问一些在校大学生和培训班的学员是否知道为什么要避免使用全局变量，他们认为一个文件中（甚至是程序中）的任何函数都可能不小心地（甚至是恶意地）修改了全局变量的值，而这样的错误源是很难发现的。有些人补充说问题的核心是访问全局变量的函数列表不清晰，这意味着出错的问题可能来自程序的任何地方。

所有这些理由都可能正确（但是本书仍然对未授权访问的重要性表示怀疑），但是滥用全局变量的问题主要是隐式耦合度。使用隐式耦合度强迫开发人员和维护人员要研究大量的代码段，才能理解程序的数据流——即哪些函数设置这些变量的值，哪些函数又使用这些变量值。而如果通过参数使用显式耦合度，我们只要研究服务器函数的函数头（或者函数原型）就可以理解数据流。这就完全不同了。

提示 尽量避免通过全局变量使用隐式耦合度。应该通过参数传递使用显式耦合度，这样维护人员（和调用函数的客户端代码程序员）可以只研究函数头就能理解函数接口，而不需要研究函数的整个代码及其调用者。

当然，我们用参数传递取代全局变量的模式来使用隐式耦合度，这并不会自动地减少代码的复杂性。我们还要选择正确的参数模式。例如，考虑下面这个版本的服务器函数

isLeap()。

```
void isLeap(int &year, int &remainder, bool &leap)     // parameters
  { if (remainder != 0)
      leap = false;
    else if (year%100==0 && year%400!=0)
      leap = false;
    else
      leap = true; }
```

这个程序语法正确吗？是正确的。那么语义正确吗？也正确。如果我们用这段代码代替程序8-2中的函数isLeap()，执行结果会是一样的吗？对任何输入来说两个函数的结果都是一样的。

但从软件质量角度来说这个函数不能算是好函数。所有的参数都按引用传递，这很容易误导维护人员以为所有3个参数都会由该函数设置，并在函数的客户使用。为了确认事实是否如此，维护人员必须研究整个服务器函数。这比只使用全局变量要好一些，因为使用全局变量需要研究服务器和客户代码。但还是远远不如程序8-2中只研究服务器函数头简单。

这个版本的函数按引用传递所有的参数，函数的开发人员就没有正确地表达他/她在设计时的想法。开发人员知道参数leap只是用作输出变量，但是没有在代码本身表达这个信息。

维护人员应该认为按引用传递意味着函数要改变参数值（除非有const修饰），而按值引用证明参数不会修改。否则维护人员就必须退回到原来的状态，即研究服务器和客户的所有细节，而不是只研究服务器函数的参数列表，于是显式耦合度的优点就消失了。

因此，我们在第7章总结的参数传递的规则就显得十分重要。如果遵循这个规则维护人员可以一致性地描述函数接口，就不用一次性地研究多个函数，也降低了需要研究的代码量。使用const修饰符就证明参数是输入参数；没有使用const修饰符则证明参数被函数修改了。注意，确实要使用这强有力的方法提高代码质量。

既然使用参数传递比使用全局变量好得多，为什么程序员还要使用全局变量呢？原因有三个：

首先是提高程序性能。使用参数的函数需要花时间为参数分配和回收内存，并复制它们的值（或者它们的地址值）。使用全局变量的函数节省了这些时间。如果是出于这个目的而使用全局变量，一定要提前确定两个问题。第一是应该很清楚程序的确有性能问题，第二是很清楚使用全局变量代替参数可以解决这个性能问题。在这里，我们强调的是清楚地知道存在的问题且使用全局变量可以解决这个问题，而不是指认为使用全局变量可能会加快程序的执行速度。

让那些并不频繁调用的函数使用全局变量并不能提高程序的执行速度。让那些需要进行外部数据输入输出的函数使用全局变量也不能提高程序的执行速度。把全局变量用在一些短小简单的函数上可能会提高这个函数的执行速度，而不会提高整个程序的执行速度，因为这些函数对程序整体性能影响不大。我们并不是说绝对不要使用全局变量，而是指出我们应该知道使用全局变量是否真的能提高程序的执行速度。

使用全局变量的第二个原因是加快开发人员编程速度。我们编写一个使用全局变量的服务器函数要比使用参数的服务器函数容易且快一些。如果使用参数传递，像程序8-2那样，我们可能提供了并不需要的额外参数，或者提供的参数不够而不得不退回去重新编写函数。编写使用参数的函数需要我们在前期规划中投入时间。

在程序8-1中，我们将变量定义为全局变量，并在函数需要的地方使用它们，事先也不必进行规划。在软件发展的初期认为这是很重要的优点，那时我们认为提高代码的编写速度是很有益的。但现在我们不再认为让代码易于编写可以节省时间和金钱，而是认为让代码易于理解才能节省时间和金钱。现在的高级语言，如C++等，都要求我们花更多的时间编写可读性更好的代码。

使用全局变量来进行函数间通信的第三个原因是开发人员之间缺乏必要的了解。他们没有考虑到在服务器函数中使用全局变量的复杂性，而只是去用它们。他们增加了和其他开发人员的交流合作，但是并没有考虑到这些交流合作会影响整个程序的质量。

我们正在解释的问题在程序设计的书中很少讨论，有些论题会出现在软件工程的书中，但这一类的书通常都是说明一般性原则，而不涉及针对某一特定语言的特定代码模式。希望这里的讨论和第7章的讨论可以说服大家养成下面这些好习惯：

- 尽量使用参数，避免使用全局变量
- 按值传递简单的输入参数，按引用传递输出参数
- 按引用传递结构类型和类类型参数，对输入参数使用const修饰符。
- 使用const修饰符传递输入数组（输出数组不需要使用const）。

警告 传递参数应该尽量遵循本书所体现的指导思想。背离这些指导思想可能会使代码编写起来更快，但不能把我们在编写函数代码时的想法准确地反映给维护人员，也就是说，维护人员不知道哪些参数用于函数输入，哪些参数用于输出。

8.2.3 如何降低耦合度

客户和服务器之间数据流的值的个数反映了耦合度的高低。值的个数越多、客户函数和服务器函数之间的依赖程度就越高，就越难做到只研究一个函数而不用研究另一个函数。

我们应该如何去减少函数间的数据流呢？这不是一个简单的任务。降低函数间依赖性的惟一办法就是重新设计，即改变函数间的责任划分。其他任何的方法都是徒劳的。

例如，有些程序员认为把参数合并为一个结构可以减少参数个数。这也有些道理，的确能减少参数传递的个数，但并不一定能降低耦合度。程序8-3显示了使用这种办法的isLeap()函数，它把3个参数合并在一个结构类型中。

程序8-3 将参数合并成结构的例子

```
#include <iostream>
using namespace std;
struct YearData
{ int year, remainder;
  bool leap; } ;

void isLeap(YearData &data)                    // one parameter only
  { if (data.remainder != 0)
      data.leap = false;
    else if (data.year%100==0 && data.year%400!=0)
      data.leap = false;
    else
      data.leap = true; }
int main()
{ YearData data;                               // local variable
```

```
    cout << "Enter the year:  ";
    cin >> data.year;                          // input fields are set
    data.remainder = data.year % 4;
    isLeap(data);                              // output field is set
    if (data.leap)                             // output field is used
          cout << data.year << " is a leap year\n";
    else
          cout << data.year << " is not a leap year\n";
    return 0;
    }
```

的确，这里的参数个数比程序8-2的少。但函数之间的数据流减少了吗？图8-3显示了这个版本程序的数据流。可以看到，仍然有两个输入值：data.year和data.remainder，也仍然有一个输出值data.leap。

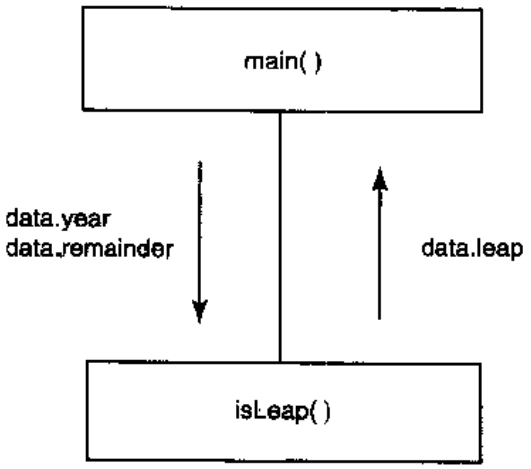


图8-3　程序8-3的对象图和数据流

我们甚至可以认为这个版本的程序更难编写，而且也肯定更难理解和重用，因为如果没有结构类型YearData，这个版本的isLeap()将无法使用。不过，这个例子的要点是为了说明程序8-3版本中的isLeap()函数及其客户之间的耦合度没有降低。这是很自然的，因为我们没有进行任何重新设计就编写了现在这个版本——这个版本的程序和程序8-2一样，在main()和isLeap()之间采用了相同的职责分配方法。因此函数之间的数据流是一样的。

有些程序员通过避免定义输出参数来降低耦合度。他们认为使用输出参数比使用函数返回值要低级。这也有一定的道理。程序8-4是另一个版本的程序，其中的isLeap()返回了一个值，而不是设置输出参数leap的值。

程序8-4　使用返回值而不是输出参数的例子

```
#include <iostream>
 using namespace std;
bool isLeap(int year, int remainder)              // fewer parameters
  { if (remainder != 0)
      return false;
    else if (year%100==0 && year%400!=0)
      return false;
    else
      return true; }
```

```
int main()
{ int year, remainder;                          // local input variables
  bool leap;                                    // local output variable
  cout << "Enter the year:  ";
  cin >> year;                                  // input variables are set
  remainder = year % 4;
  leap = isLeap(year,remainder);                // output variable is set
  if (leap)                                     // output variable is used
        cout << year << " is a leap year\n";
  else
        cout << year << " is not a leap year\n";
  return 0;
  }
```

在程序8-4中，数据流中的参数个数也比程序8-2的参数个数少。isLeap()函数也更容易编写，因为不必处理引用类型的参数。而且使用起来也更容易。例如我们可以不使用变量leap，而是在main()的if语句中直接使用isLeap()的返回值，而不用先设置局部变量的值。

```
int main()
{ int year, remainder;                      // no variable leap
  cout << "Enter the year:  ";
  cin >> year;                              // input variables are set
  remainder = year % 4;
  if (isLeap(year,remainder)==true          // output value is used
    cout << year << " is a leap year\n";
  else
    cout << year << " is not a leap year\n";
  return 0; }
```

main()函数和isLeap()函数之间的数据流减少了吗？实际上没有。图8-4显示了这个版本程序的数据流。在图中我们可以看到仍然有两个输入变量year和remainder，以及一个由函数返回值表示的输出值。

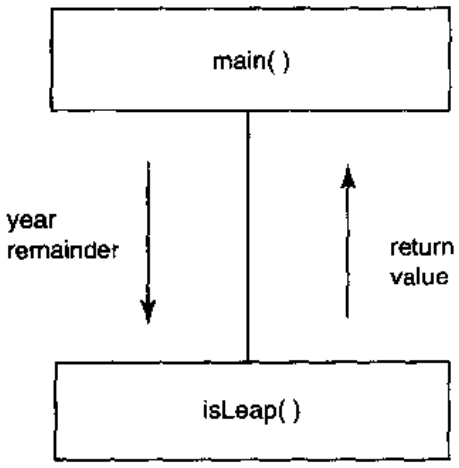


图8-4 程序8-4的对象图和数据流

因为我们没有进行重新设计，因此耦合度也没有降低，本程序在main()函数和isLeap()函数之间划分职责的方式和程序8-2采用的方式一样。

为了降低耦合度，我们必须认真分析计算性职责的分配方式，并应用本章开始时列举的

原则。实现这个目标的一个方法是，在数据流中找出哪些本来应该属于一起的成分被拆分开了，将本来应该属于一个函数的计算拆分开来，常常会导致那些被分开的计算彼此间要进行通信。当计算过程放在几个函数中实现时，函数间的通信往往表现为多余的数据流。通过将拆分开来的几个函数统一为一个函数，我们可以消除函数间的通信。

将本来属于一起的代码拆分开来所造成的问题之一是：如果只研究服务器函数代码而不研究客户函数代码，就不能清楚地知道参数的意义。如在程序8-4中，如果只是研究函数isLeap()就不能推导出参数remainder的意义。维护人员只有在研究了客户函数main()之后，才知道这个变量代表年被4除之后的余数。这个值在main()中只是用来作为给isLeap()的参数。因此，将remainder的计算及其使用合并到同一个函数中（本例中是isLeap()函数）是很有意义的。

程序8-5显示了重新设计后的代码版本，它将remainder的计算从函数main()中移到了服务器函数isLeap()中。图8-5显示了两个函数之间的数据流。

程序8-5　从客户把职责推向服务器的例子

```
#include <iostream>
using namespace std;

bool isLeap(int year)                    // even fewer parameters
   { int remainder=year%4;               // do not separate what belongs together
     if (remainder != 0)
       return false;
     else if (year%100==0 && year%400!=0)
       return false;
     else
       return true; }

int main()
{ int year; // local data - no remainder
  cout << "Enter the year:  ";
  cin >> year;                           // input variable is set
  if (isLeap(year))                      // output variable is used
    cout << year << " is a leap year\n";
  else
    cout << year << " is not a leap year\n";
  return 0;
  }
```

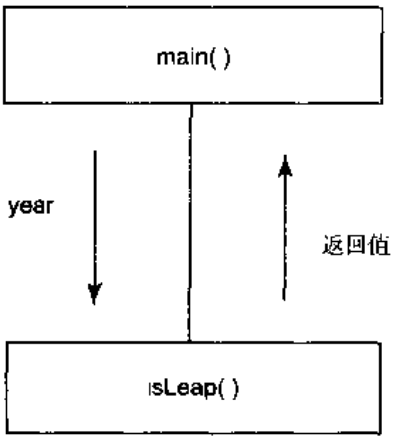


图8-5　程序8-5的结构图和数据流

实际上，现在的isLeap()只需要从main()获得一个值，因为它自己计算remainder，而不需要麻烦其客户函数在调用它之前计算remainder。

把remainder的有关计算从一个函数移到另一个函数就叫做重新设计，因为我们修改了函数之间的职责推向分配。注意我们是通过将职责从客户函数移到服务器函数来将已经分开的代码合并的。这也是一个将职责推向服务器的例子。这种做法虽然不一定总是有益，但常常很有用。

这是很有用的技巧。降低函数间的通信可以加速维护，方便重用，并在函数被不同的程序员编写时（或者在同一个程序员在不同阶段开发时）降低程序员之间的通信。一定要常常检查，是否将应该属于一起的代码分开成代码碎块。

我们也应该时时考虑函数间进行过度通信的危险性。降低耦合度的最好方法是，通过将应该属于一起的代码合并在一起以消除通信的必要性。

采用这种方法可以到什么程度呢？如果将用户提示和变量year的定义都放到isLeap()中有意义吗？这样可以进一步降低函数间的数据流。然而，这样就需要程序员之间就用户界面进行交流——即哪个函数应该负责用户界面呢？这样做的确可以降低函数isLeap()的耦合度，因为它把计算和输入/输出合并在一起了。

在程序8-5中，main()函数负责用户界面，isLeap()负责进行有关计算。将用户界面分开和将计算分开一样有害。一定要确保任何函数的责任域都清晰地定义。

程序8-5还可以进一步改进，例如不使用remainder变量。

```
bool isLeap(int year)
  { if (year % 4 || year%100==0 && year%400!=0)
      return false;
    else
      return true; }
```

如果对紧凑的代码特别有兴趣，也可以使用下面的方式。

```
bool isLeap(int year)
  { return (year % 4 || year%100==0 && year%400) }
```

正如我们在第4章提到的，这种改进是否值得还没有定论。无论如何，它们都不会影响耦合度，因为它们没有改变函数之间的工作分配。

警告 通常，当开发人员添加不同的函数时，耦合度增加；因为这些操作应该在相同的函数中实现。这就增加了开发人员之间的交流，阻碍了维护和重用。一定要时时考虑这些危险性。

8.3 数据封装

使用C++和使用其他的高级语言一样，程序员把算法的复杂性隐藏在函数中。每一个函数都是为直接实现某一特定目标而编写的语句集合。函数名通常反映了这个目标。一般来说，函数名由两部分组成：描述行为的行为动词和描述行为对象（或者行为主语）的名词，例如processTransaction()、acceptInput()等。当行为的对象在上下文中很明显时（例如作为参数传送给函数），我们也可以只使用动词，如add()、delete()等。

在函数体内的语句集合中可以包含简单的赋值运算、复杂的控制结构、或者对其他函数的调用。其他函数可以是标准的库函数，也可以是为这个特定的工程而度身定做的程序员定义函数。

从程序员的角度来看，库函数和程序员定义函数之间的差别只在于：程序员定义函数的实现源代码是可供检查的；而库函数的源代码是无法获得的。即使有库函数的源代码，客户端代码程序员也不会花费额外的精力去查看这些代码。他们想知道的只是服务器函数界面的描述：即调用函数的代码应该提供什么参数，函数对哪些值进行计算，输出值与输入值的关系如何，有哪些限制和异常处理等。这些让程序员可以选择合适的库函数，并正确使用它们。

而程序员定义函数则通常不是现有的函数，而是需要重新设计的。常常修改这些函数的源代码以更好地适应客户函数的需要。这些函数不像库函数那样经过严格的测试。当出现问题时，可能是客户函数或者服务器函数的错误。因此客户端代码程序员（维护人员）必须一起研究相关函数——客户和服务器——的源代码。这就让客户端代码程序员（维护人员）的任务比使用库函数时繁重。我们都希望设计程序员定义的函数让程序的复杂性降低。数据封装概念就是帮助程序员达到这个目标的概念之一。在服务器函数经过严格的测试后，客户端代码程序员（维护人员）可以把它们看成库函数对待，就像处理有指定界面的黑盒一样。

让我们考虑一个简单的例子：处理几何图形（例如圆柱体）的图形包的一部分。为了简单起见，我们假设每个圆柱体对象都只有两个double类型的特征，即圆柱体的半径和高。

```
struct Cylinder {
  double radius, height; } ;
```

程序将会提示用户输入两个圆柱体的半径和高。如果第一个圆柱体的体积比第二个的小，程序将会把第一个圆柱体放大，各维都会放大20%，并输出改变大小后的半径和高。在现实生活中，这一段代码可能是某个程序的一部分，该程序或者使用圆柱体对象描述化学反应堆的热交换，或者研究微芯片中的电流情况，或者分析钢铁建筑框架。本例子的实现代码如程序8-6所示。图8-6是这个例子的运行结果。

程序8-6　直接访问底层数据表示的例子

```
#include <iostream>                                    // no encapsulation yet
using namespace std;

struct Cylinder {                                      // data structure to access
  double radius, height; } ;

int main()
{
  Cylinder c1, c2;                                     // program data
  cout << "Enter radius and height of the first cylinder:  ";
  cin >> c1.radius >> c1.height;                       // initialize first cylinder
  cout << "Enter radius and height of the second cylinder: ";
  cin >> c2.radius >> c2.height;                       // initialize second cylinder
  if (c1.height*c1.radius*c1.radius*3.141593           // compare volumes
    < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2;  c1.height *= 1.2;               // scale it up and
    cout << "\nFirst cylinder changed size\n";         // print new size
    cout <<"radius: "<<c1.radius<<" height: "<<c1.height<<endl; }
  else                                                 // otherwise do nothing
    cout << "\nNo change in first cylinder size" << endl;
  return 0;
  }
```

在这段代码里，main()函数直接访问Cylinder的数据表示，而不需要任何服务器函数的帮助。因此，它将数据访问（例如源代码中的c1.radius）和数据操纵（例如计算体积、

放缩大小、打印圆柱体数据）混合在一起。于是维护人员必须在代码中分析出操作的意义，而不是通过服务器函数名辨认出来。

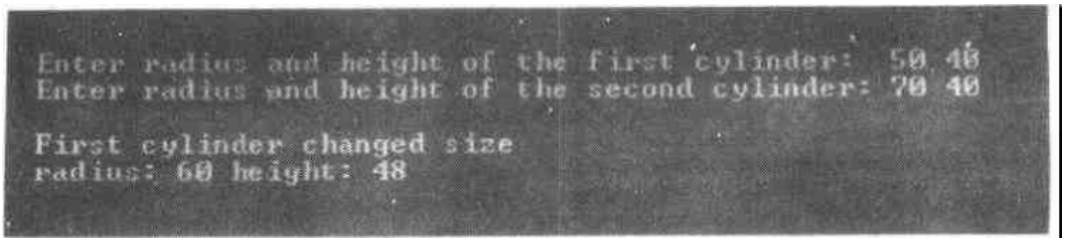


图8-6 程序8-6的输出结果

当然，代码设计人员可以提供注释解释代码的意义，就像程序8-6一样。然而，其他人写的注释对读者来说常常不够清晰，或者没有注释；或者更为糟糕的是，当源代码更新时设计人员没有来得及更新注释。

解决这个问题的一个方法是找到服务器函数集合，这些函数代表客户代码访问Cylinder结构的域。通过将进行计算的职责推向服务器函数，我们可以将客户代码从计算的底层细节中解脱出来。而计算的高层意义仍然保留在被客户代码调用的服务器函数的函数名中。这样，客户代码就变成自解释性了：即客户代码的读者理解客户函数完成的工作，即使他还不清楚服务器函数完成这个工作的细节情况。

```
int main()                                  // pushing responsibility to servers
{
   Cylinder c1, c2;                         // program data
   enterData(c1,"first");                   // initialize first cylinder
   enterData(c2,"second");                  // initialize second cylinder
   if (getVolume(c1) < getVolume(c2))       // compare volumes
      { scaleCylinder(c1,1.2);              // scale it up and
        printCylinder(c1); }                // print new size
   else                                     // otherwise do nothing
        cout << "No change in first cylinder size" << endl;
   return 0;
   }
```

要理解这一版本的main()、理解服务器函数enterData()、getVolume()、scaleCylinder()和printCylinder()是如何实现其功能的并不重要。这个客户代码的注释和没有使用访问函数的程序8-6的注释一样。但和程序8-6不同的是，这些注释行不再有什么用。它们只是重复了服务器函数名在被客户代码调用时的意义。这就是“将职责从客户代码推向服务器函数”的一个重要优点，也是我们在本章开始的时候指出的一个重要原则。

根据传统的编程方法，注释行是很重要的。如果一个程序没有注释，将会要求开发人员增加它们。有了数据封装，当计算的细节推到服务器函数时，客户代码就不再需要注释行；因为从函数调用中的函数名可以清楚地知道处理的意义。如果没有注释行的客户代码意义仍然模糊，那就意味着服务器函数设计得不好——应该要求程序员重新设计代码（而不是添加注释）。

将数据访问和数据值运算混杂在一起的另一个编码风格问题是，数据的验证会变得模糊和糟糕，甚至常常被省略。例如，代码的第一个版本（程序8-6）没有进行任何的数据验证。在本例中，输入数据由用户提供，程序必须提供对用户输入错误的容错性处理。而现实生活中，输入数据可能来自外部文件或者通信行，这些数据源也可能像人类用户那样产生破坏性错误。但是，即使是最严格的错误恢复形式也会让客户代码意义模糊。例如将圆柱体的域设

置为缺省值。

```
int main()
{ Cylinder c1, c2;                                    // program data
  cout << "Enter radius and height of the first cylinder:  ";
  cin >> c1.radius >> c1.height;                      // initialize first cylinder
  if (c1.radius < 0) c1.radius = 10;                  // defaults for corrupted data
  if (c1.height < 0) c1.height = 20;
  cout << "Enter radius and height of the second cylinder: ";
  cin >> c2.radius >> c2.height;                      // initialize second cylinder
  if (c2.radius < 0) c2.radius = 10;                  // defaults for corrupted data
  if (c2.height < 0) c2.height = 20;
  if (c1.height*c1.radius*c1.radius*3.141593          // compare volumes
    < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2;  c1.height *= 1.2;              // scale it up and
    cout << "\nFirst cylinder changed size\n";        // print new size
    cout <<"radius: "<<c1.radius<<" height: "<<c1.height<<endl; }
  else                                                // otherwise do nothing
    cout << "\nNo change in first cylinder size" << endl;
  return 0;
  }
```

使用访问函数可以从客户代码中消除底层的数据验证。例如可以使用validateCylinder()函数，如果输入数据值为负数它就将圆柱体的域设置为缺省值。程序8-7显示了这个版本的程序，其输出结果与程序8-6的版本结果相同。

程序8-7　使用访问函数将客户代码从数据域名分隔开的例子

```
#include <iostream>                    // encapsulation with server functions
using namespace std;

struct Cylinder {                      // data structure to access
    double radius, height; } ;

void enterData(Cylinder &c, char number[])
{ cout << "Enter radius and height of the ";
  cout << number << " cylinder:  ";
  cin >> c.radius >> c.height; }       // initialize cylinder

void validateCylinder(Cylinder c)
{ if (c.radius < 0) c.radius = 10;   // defaults for corrupted data
  if (c.height < 0) c.height = 20; }

double getVolume(const Cylinder& c)  // compute volume
{ return c.height * c.radius * c.radius * 3.141593; }

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor;  c.height *= factor; }     // scale dimensions

void printCylinder(const Cylinder &c)   // print object state
{ cout << "radius: " <<c.radius << " height: " <<c.height <<endl; }

int main()                             // pushing responsibility to server functions
{
  Cylinder c1, c2;                     // program data
  enterData(c1,"first");               // initialize first cylinder
  validateCylinder(c1);                // defaults for corrupted data
  enterData(c2,"second");              // initialize second cylinder
```

```
validateCylinder(c2);                  // defaults for corrupted data
if (getVolume(c1) < getVolume(c2))     // compare volumes
  { scaleCylinder(c1,1.2);             // scale it up and
    cout << "\nFirst cylinder changed size\n";       // print size
  printCylinder(c1); }
else                                   // otherwise do nothing
  cout << "No change in first cylinder size" << endl;
return 0;
}
```

我们可以发现，新的程序设计方法实际上产生了更多的源代码。如果是一个实时系统，额外的函数调用将会影响程序性能。使用内联函数就可以消除这个问题。

这样编写代码的好处是产生两个不同的问题关注域：一个关注域是程序员定义类型Cylinder及其访问函数的设计；另一个关注域则与使用Cylinder对象和调用Cylinder访问函数的客户代码相关。如果使用传统的程序设计（见程序8-6），就不存在不同的问题关注域。如果程序员定义类型Cylinder结构的域名改变了，整个程序的代码都必须被查看，因为在程序的任何地方都可能使用这些域名。如果使用新的程序设计方法（如程序8-7），对Cylinder域名的修改只会影响访问函数——它们是定义良好的函数集，而不论整个程序有多大，程序的其他部分都不会受到影响。图8-7以对象图的形式显示了客户代码和服务器代码之间的这种关系。客户函数main()调用访问Cylinder对象域的服务器函数。从客户函数的角度来看，服务器函数将Cylinder类的设计封装起来。

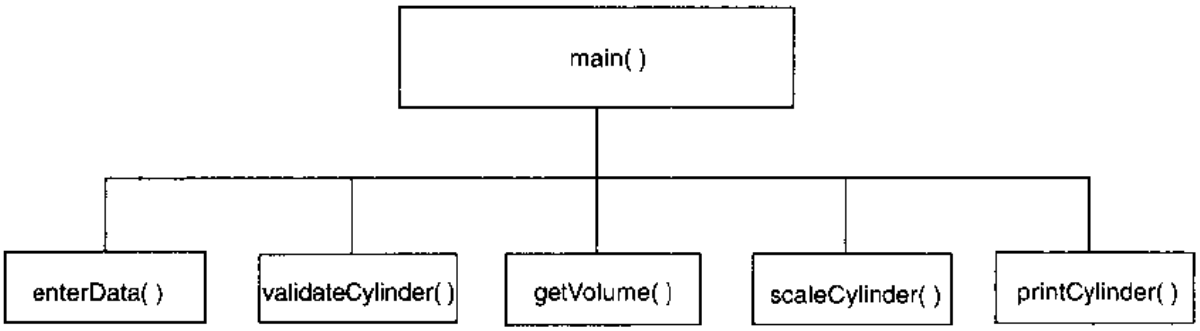


图8-7 程序8-7的对象图

数据封装是一个较新的概念，还没有很正确地理解。许多程序员认为数据封装就是使用函数来保护数据不会被错误地或者未授权地修改。如果不使用数据封装，客户代码就可以通过按名字直接访问数据域来随意地不被人随意地修改数据。使用数据封装，客户代码会调用访问函数，例如函数scaleCylinder()等，利用这些访问函数修改数据域。

对数据保护的关注类似于对使用全局变量带来危害的关注：如果一个全局变量对整个程序有效，有些人就可能错误地设置了这些变量的值而损害程序的其他部分。类似地，如果数据域名对整个程序有效，会发生同样的可能性。传递参数保护了全局变量，数据封装保护了数据域。

这种数据保护的观念在一代又一代的程序员中流传，因为它简单且好像很有道理，而且接受这种观念比反对要容易一些。而我们现在则有异议，数据保护的确有些道理，但其重要性比较小。数据封装的真正意义在于程序组成部分的可读性和独立性，这才是本章的核心主题。

实际上，参数传递并不能保护数据。如果程序员错误地认为某个变量的值需要改变，他可以直接对变量赋值（如果是全局变量），也可以对函数的参数赋值（如果是按引用调用或者是按指针传递参数）。类似地，如果程序员错误地认为c1.radius的值要改变，可以使用直接赋值（如果没有使用数据封装），或者通过调用某个访问函数，如setCylinder()等（如果使用了数据封装）。两者之间并没有什么不同。

真正的解释是我们在本章开头阐述的关注域分开原则。在维护期间将对客户代码和对访问函数的关注域分开，这对全局变量和数据域而言都是很重要的。如果要改变全局变量的使用和变量名，我们必须搜索所有的程序文件以确定可能的依赖性，因为任何文件都可能使用和修改了这些全局变量值。这就不是一个定义清晰和比较小的关注域，因为维护人员的注意力分散于整个程序。这是劳动力密集型和容易出错的情况。

同样，如果我们在没有封装数据的程序中修改了数据域的名字或者类型，也需要搜索所有的程序文件以确定可能的依赖性，因为任何文件都可能使用和修改了这些域值。这并不是一个定义清晰和比较小的关注域，因为维护人员的注意力分散于整个程序。

注意，我们并不是抱怨修改代码是一件很麻烦的工作。我们到底花了多少时间编写和修改代码呢？这常常只是整个程序开发过程中最容易和最短的阶段。我们抱怨的是，当Cylinder类（或者其他数据结构）的设计改变时，程序中没有任何标注清晰的部分供我们查看并进行相应修改。而是必须在所有地方查找需要进行修改的部分，并确保不会引入不想要的副作用。这就让维护工作容易出错且代价昂贵。

如果我们使用数据封装，当数据域的名字或者类型改变时，需要进行相应修改的只是访问函数集合，而程序的所有其他部分不受影响。调用访问函数的程序其他部分需要重新编译，但是它们的源代码不需要修改。因此，维护人员的注意力比较集中——限制在与域名相关的代码中。这才是数据封装的真正益处：通过不直接使用数据域名，客户代码避免了对数据设计的依赖性。

学会用数据封装的思想考虑代码设计是很重要的。如果这样做，就可以创建两个不同关注域：使用和不使用数据域名的代码段。

但是，使用访问函数本身并不一定会提供代码组件的可读性和独立性。因此我们需要另外一个判断代码质量的评价标准：信息隐藏。

8.4 信息隐藏

信息隐藏的概念也和将关注域分开的原则有关。一般来说，如果没有信息隐藏，编写代码（或者维护代码）的程序员必须同时了解两套设计决策或者两方面的知识。一个是数据的设计（如类型Cylinder），另一个是与应用程序相关的数据操纵（如设计域、比较体积和缩放等）。

使用信息隐藏可以分开关注域。编写（或者维护）客户代码的程序员只需要关心与应用程序相关的数据操纵，而不需要关心数据设计。编写（或者维护）数据访问函数的程序员只需要关心数据设计，而不用关心与应用程序相关的数据操纵。

如果觉得信息隐藏和数据封装很相似，这就对了。必须承认，我们所读到的信息隐藏定义大多是含糊不清和缺乏可操作性的，都没有严格地区分数据封装和信息隐藏，也没有说明如何判别出程序缺乏信息隐藏或者如何实现信息隐藏。大多数人把信息隐藏等同于数据封装。

其实，数据封装相对于信息隐藏来说，其范围更窄。我们只是想对客户代码封装数据域的名字和类型，这样客户代码就不会显式地使用底层数据域的名字。在我们的例子程序中，就是main()函数中不出现c1.radius和c1.height等如此直接的代码。通过访问函数实现的封装能改善代码质量，增强其可读性和各组成部分的独立性。

信息隐藏和数据封装有什么不同呢？在回答这个问题之前，我们先来看一个数据封装得不是很好的例子。这个例子试图通过引入对Cylinder对象执行操作的服务器函数实现封装，例如该函数返回Cylinder域的值或者设置Cylinder尺寸。这些服务器函数也被称为访问函数，因为它们代表客户代码访问Cylinder的数据。这里的“访问”包括两种不同的访问类型——即这些函数可以获取域的值也可以修改域的值。

```
void setRadius(Cylinder &c, double r)          // modifier function
{ c.radius = r; }

void setHeight(Cylinder &c, double h)          // modifier function
{ c.height = h; }

double getRadius(const Cylinder& c)            // selector function
{ return c.radius; }

double getHeight(const Cylinder& c)            // selector function
{ return c.height; }
```

main()函数不必使用Cylinder组成成员的名字，如果这些名字改变了，需要进行相应修改的也只是setRadius()、setHeight()、getRadius()和getHeight()，而main()或者Cylinder的其他任何客户都不需要修改。程序8-8显示了这些访问函数的使用。该程序的输出和程序8-6的输出一样——因为我们修改了代码的设计而没有修改代码的功能。

程序8-8 效率低的数据封装的例子

```
#include <iostream>                            // awkward encapsulation
using namespace std;

struct Cylinder {                              // data structure to access
  double radius, height; } ;

void setRadius(Cylinder &c, double r)          // modifier
{ c.radius = r; }

void setHeight(Cylinder &c, double h)          // modifier
{ c.height = h; }

double getRadius(const Cylinder& c)            // accessor
{ return c.radius; }

double getHeight(const Cylinder& c)            // accessor
{ return c.height; }

int main()
{
Cylinder c1, c2;  double radius, height;                  // program data
cout << "Enter radius and height of the first cylinder:  ";
cin >> radius >> height;                                  // initialize data
setRadius(c1,radius); setHeight(c1,height);
```

```
if (getRadius(c1)<0) setRadius(c1,10);                    // verify data
if (getHeight(c1)<0) setHeight(c1,20);
cout << "Enter radius and height of the second cylinder: ";
cin >> radius >> height;                                  // initialize data
setRadius(c2,radius); setHeight(c2,height);
if (getRadius(c2)<0) setRadius(c2,10);                    // verify data
if (getHeight(c2)<0) setHeight(c2,20);
if (getHeight(c1)*getRadius(c1)*getRadius(c1)*3.141593
  < getHeight(c2)*getRadius(c2)*getRadius(c2)*3.141593)
{ setRadius(c1,getRadius(c1)*1.2);
  setHeight(c1,getHeight(c1)*1.2);                        // scale up
  cout << "\nFirst cylinder changed size\n";               // print new size
  cout <<"radius: "<<c1.radius<<" height: "<<c1.height<<endl; }
else                                                      // otherwise do nothing
  cout << "No change in first cylinder size" << endl;
return 0;
}
```

对main()函数而言，Cylinder的数据域名的确已封装起来。如果在重新设计过程中修改了这些域名，只有有限的而且容易辨认的访问函数集需要修改。即使整个程序很庞大，也不需要对程序其他任何地方进行修改甚至检查。虽然需要重新编译，但这并不困难。图8-8显示了这个设计的对象图。和我们在第1章介绍的对象图1-7相似，这个对象图显示了服务器函数setRadius()、setHeight()、getRadius()、getHeight()都是概念上属于一起的。它们代表客户代码访问Cylinder结构的域radius和height。该客户代码仅仅通过客户的访问函数存取服务器的数据，而不是直接地存取数据。

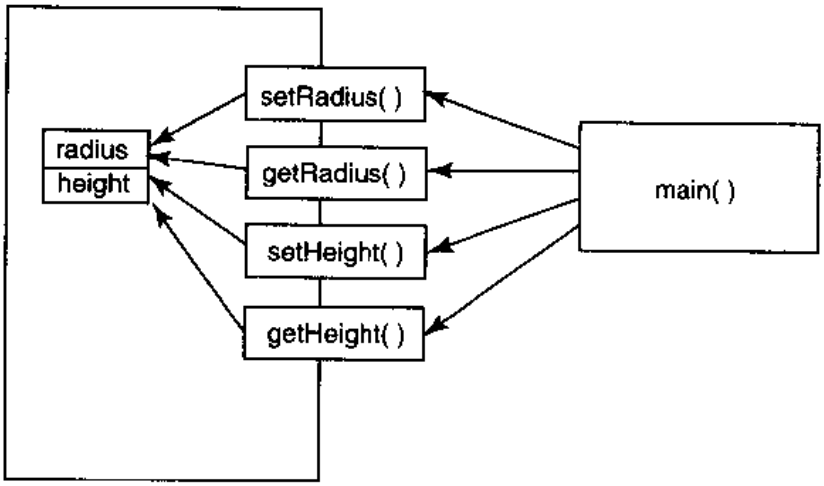


图8-8　程序8-8的对象图

然而，这样的数据很糟糕，实际上可以说没有什么用，这个设计并没有使用在本章开头列举的设计原则。访问函数为实现客户目标几乎没有做什么事情：因为数据操纵的任务没有被推到服务器函数，而是保留在客户。虽然使用了访问函数，客户代码main()仍然混杂着对数据的访问（例如调用getRadius()）和数据操纵，这样仍然不容易把握计算（例如计算体积、缩放大小等）的意义。如果程序员定义类型Cylinder的域数目改变，访问函数的个数就会进行相应改变，客户的代码也必须进行修改。

为了正确地选择访问服务器函数集，必须考虑客户代码的职责。在本例中，客户代码负

责初始化圆柱体数据、验证对象数据、计算圆柱体体积、缩放圆柱体大小、显示圆柱体数据。因此我们就应该对应地实现如下访问函数：setCylinder()、validateCylinder()、getVolume()、scaleCylinder()、printCylinder()等。

有了这些函数，我们就将职责从客户代码推向服务器代码。现在是由服务器函数负责初始化圆柱体数据、验证圆柱体数据、计算圆柱体体积、缩放圆柱体大小、显示圆柱体数据。客户代码只是提出操作请求。因此main()的操作就可以表示成对服务器函数的调用。

采用上述办法，数据访问和数据操纵的混杂就不存在了。客户代码指定应该做什么（设置数据域、计算体积等），而服务器代码指定如何做。Cylinder的数据表示已封装起来，如果其域名改变了，并不会影响客户代码；如果Cylinder增加了更多的域，客户代码也不受影响。（如果要完全实现这个效果，还需要封装输入操作。）

客户设计人员和服务器设计人员共享的信息就限制于服务器函数的名字和界面。客户端代码程序员和服务器程序员的关注域也分开了，前者注意与应用程序相关的高层操作，后者只用关系数据域名和底层计算。

即使是如此小的一个例子，我们也可以从中体会到使用访问函数的好处。客户代码被表示成意义更明显的与应用程序相关的操作。而程序8-6中的c1.height*c1.radius*c1.radius*3.141593到底有什么意义，维护人员必须察看代码才能推导出来。表达式c1.radius*=1.2;和c1.height*=1.2;也是一样，我们必须认真琢磨，是不是圆柱体的三维都放大？放大的比例是不是三维都相同？而在打印语句中，我们也要研究，是显示了圆柱体三维数据还是只显示了一部分。当数据访问和与应用程序相关的操作交织时，更难分析出程序处理的意义。

使用访问函数也可以让用户输入数据的验证工作更加简单——因为main()函数中不会再充斥着数据验证的细节。如果数据表示（圆柱体设计或者域名）改变了，需要相应改变的也只是服务器函数。正如我们前面说明的，这不仅仅是维护的工作量问题，而是注意力的范围问题。如果没有访问函数，可能修改的部分是整个程序（因为任何地方都可能使用Cylinder）；如果使用了访问函数，需要修改的部分就很容易辨认——即访问Cylinder数据表示的访问函数。

这种方法也可以增加代码的重用性。如果不使用访问函数，任何使用Cylinder对象的算法必须从头开始编写和进行验证。如果使用访问函数，新的算法就可以表示成对这些函数的调用，这样所有的操作都只需要验证一次。

该方法的缺点就是必须编写和测试更多的源代码，但有些人会认为这实际上是另外一个优点。从软件的整个开发周期来说，输入代码只占很小一部分时间，而所有其他开发步骤需要阅读代码——包括调试、测试、集成和维护。将客户代码表示成对访问函数（已经被编写和经过测试的）的函数调用，可以让这些开发步骤更加容易、错误更少和开销更小。

那么，信息隐藏的准则比数据封装究竟多了些什么呢？我们再来分析服务器函数validateCylinder()和getVolume()。前一个函数封装了合法性验证的操作、缺省值等。这是很好的，因为客户代码不需要了解合法性验证的细节，而只是需要知道进行了验证就可以了。第二个函数封装了几何计算。这也是很好的，因为客户代码不需要关心几何规则，而只需要知道计算了圆柱体体积就足够了。

但这两个函数从信息隐藏的角度来看设计得都不好。它要求客户设计人员需要知道服务

器设计人员知道的信息，扩展了客户设计人员的关注域，在客户代码中引入了数据操作的信息而不是在服务器函数完成计算。

第一个函数validateCylinder()暴露了进行数据合法性验证的需求，而这不应该属于客户代码设计人员和维护人员的关心范围。解决这个问题的方法是重新设计，即改变函数列表及其职责。一个好的重新设计方案是把validateCylinder()和enterData()函数合并起来。

```
void enterData(Cylinder &c, char number[])
{ cout << "Enter radius and height of the ";
  cout << number << " cylinder:  ";
  cin >> c.radius >> c.height;          // initialize cylinder
  if (c.radius < 0) c.radius = 10;      // defaults for corrupted data
  if (c.height < 0) c.height = 20; }
```

我们已经反复多次说过，内聚性、耦合度、数据封装、信息隐藏等软件质量评估准则缺乏可操作性，它们只是指出设计上存在的缺点，但是没有具体地指导我们如何修改设计以消除这些缺点。而本章开头列出的原则是可操作性的，指示了我们应该怎样修改代码。在本例中，通过将职责推向服务器函数来提高信息隐藏性。该设计不再强迫客户函数调用两个分开的服务器函数enterData()和validateCylinder()，而是调用一个访问函数就行了。

getVolume()函数同样违反了将职责推到服务器函数的原则，也让客户代码了解了不需要知道的信息。客户代码需要知道的是一个圆柱体是否比另一个圆柱体大，但服务器函数没有响应这个需求，而是返回圆柱体的体积值，让客户代码进行相应的操作。圆柱体体积的信息应该对客户代码来说是隐藏的。因此，为了满足客户代码的需求，我们应该修改设计，例如引入函数firstIsSmaller()。

```
bool firstIsSmaller(const Cylinder& c1, const Cylinder& c2)
{ if (c1.height*c1.radius*c1.radius*3.141593      // compare volumes
    < c2.height*c2.radius*c2.radius*3.141593)
    return true;
  else
    return false; }
```

程序8-9显示了修改后的最终版本，它采用了合理的数据封装和信息隐藏。注意到代码的功能和以前所有版本的程序完全一样。我们修改的只是程序设计，而影响代码质量的也是设计。程序8-9的输出结果和程序8-6的输出结果一样。

程序8-9　采用了数据封装和信息隐藏

```
#include <iostream>
using namespace std;

struct Cylinder {                                   // data structure to access
    double radius, height; } ;

void enterData(Cylinder &c, char number[])
{ cout << "Enter radius and height of the ";
  cout << number << " cylinder:  ";
  cin >> c.radius >> c.height;                      // initialize cylinder
  if (c.radius < 0) c.radius = 10;                  // defaults for corrupted data
  if (c.height < 0) c.height = 20; }

bool firstIsSmaller(const Cylinder& c1, const Cylinder& c2)
```

```
{ if (c1.height*c1.radius*c1.radius*3.141593      // compare volumes
       < c2.height*c2.radius*c2.radius*3.141593)
    return true;
  else
    return false; }

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor;  c.height *= factor; }     // scale dimensions

void printCylinder(const Cylinder &c)            // print object state
{ cout << "radius: " <<c.radius << ' height: " <<c.height <<endl; }

int main()                          // pushing responsibility to server functions
{
  Cylinder c1, c2;                                // program data
  enterData(c1,"first");                          // initialize first cylinder
  enterData(c2,"second");                         // initialize second cylinder
  if (firstIsSmaller(c1,c2))
    { scaleCylinder(c1,1.2);                      // scale it up and
      cout << "\nFirst cylinder changed size\n";  // print new size
      printCylinder(c1); }
  else                                            // otherwise do nothing
      cout << "\nNo change in first cylinder size" << endl;
  return 0;
  }
```

图8-9显示了程序8-9的结构。与前一个对象图相似，函数enterData()、firstIsSmaller()、scaleCylinder()、printCylinder()属于一起。这里的服务器函数能更好地为客户代码服务，因为访问函数帮助客户代码完成工作，而不是提供信息让客户代码进行进一步的处理。

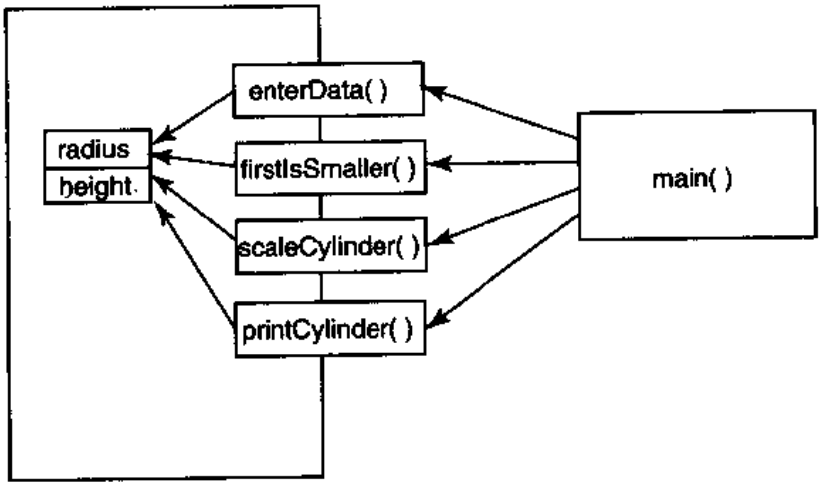


图8-9 程序8-9的对象图

8.5 一个有关封装的大型例子

我们将要讨论的程序是关于一个表达式合法性验证的问题。为了简化问题，我们只是判断输入表达式中小括号和中括号彼此是否匹配。我们考虑函数checkParen()，它按顺序

一个个地扫描存放在数组中的表达式的字符，直到碰到结束符（表达式的结尾）或者发现括号已不匹配才停下来。例如，表达式a = (x [i]+5)*y应该识别为一个合法的表达式，而表达式a = (x [i) +5]*y是不合法的。

这个示例程序使用了两个全局数组buffer[]和store[]。下标i用来指示数组buffer[]的字符，而下标idx用来指示store[]的字符。函数还会返回标志valid，它已初始化为1（真），如果在表达式验证过程中发现表达式不合法，则把valid设为0（假）。在循环结构中，checkParen()函数将会逐一地判断buffer[]数组中的每一个字符。如果当前的字符是一个左括号（小括号或者中括号），将会等到相应的右括号出现时才做出判断的结论，所以我们还要把一些字符保存到store[]数组中（并且修改数组下标idx）。

```
char buffer[81];  char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;          // initialize data
  while (buffer[i] != '\0' && valid)     // end of data or error?
    { c = buffer[i];                     // get next symbol
    if (c=='(' || c=='[')                // is next symbol left?
      { store[idx] = c;  idx++; }        // then save it away
// THE REST OF THE CODE
  return valid; }
```

如果数组buffer[]中的下一个符号是右括号（右小括号或者右中括号），代码会从数组store[]中读取最后一个符号（再次要调整数组下标idx）。此时，程序会检查两个符号是否匹配。具体说来就是如果数组buffer[]中的符号是右小括号，则数组store[]中的符号应该是左小括号；类似地，如果数组buffer[]中的符号是右中括号，则数组store[]中的符号应该是左中括号而不是左小括号。如果两个符号匹配，就不需要做什么其他操作，代码继续处理数组buffer[]中的下一个符号；如果两个符号不匹配，表达式不合法，代码设置标志valid的值为false，然后中断循环，将0值返回给客户代码。

```
char buffer[81];  char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = 1;             // initialize data
  while (buffer[i] != '\0' && valid)     // end of data or error?
    { c = buffer[i];                     // get next symbol
      if (c=='(' || c=='[')              // is next symbol left?
        { store[idx] = c;  idx++; }      // then save it away
      else if (c==')' || c==']')         // is next symbol right?
      { idx--;  sym = store[idx];        // get the last symbol
        if (!((sym=='(' && c==')') ||
              (sym=='[' && c==']')))     // if they do not match
              valid = false; }           // then it is an error
// THE REST OF THE CODE
  return valid; }
```

当然，上面的一段代码把问题看得太简单了，代码怎么知道数组store[]中一定会有一个符号和数组buffer[]中的右括号相匹配呢？如果输入表达式中有多个右括号，但没有与之相匹配的前续左括号，就会清空数组store[]，其下标idx会变成负数，这种情况下我们也应该声明表达式无效。

```
char buffer[81];  char store[81];
```

```
int checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = 1;                    // initialize data
  while (buffer[i] != '\0' && valid != 0)       // end of data or error?
  { c = buffer[i];                              // get next symbol
    if (c=='(' || c=='[')                       // is next symbol left?
      { store[idx] = c;  idx++; }               // then save it away
    else if (c==')' || c==']')                  // is next symbol right?
      if (idx > 0)                              // does saved symbol exist?
        { idx-;  sym = store[idx];              // get the last symbol
          if (!((sym=='(' && c==')') ||
                (sym=='[' && c==']')))          // if they do not match
            valid = 0; }                        // then it is an error
      else
        valid = 0;       // if no saved symbol to match, it is an error
// THE REST OF THE CODE
  return valid; }                               // return the error status
```

在上面的函数代码中，我们已经考虑了从数组buffer[]中读出的字符是左括号、右括号时，应该如何处理；如果既不是左括号又不是右括号，我们只需要继续处理数组buffer[]中的下一个字符，即把下标i的值加1。

```
char buffer[81];  char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;                 // initialize data
  while (buffer[i] != '\0' && valid)            // end of data or error?
  { c = buffer[i];                              // get next symbol
    if (c=='(' || c=='[')                       // is next symbol left?
    { store[idx] = c;  idx++; }                 // then save it away
    else if (c==')' || c==']')                  // is next symbol right?
    if (idx > 0)                                // does saved symbol exist?
    { idx-;  sym = store[idx];                  // get the last symbol
      if (!((sym=='(' && c==')') ||
            (sym=='[' && c==']')))              // if they do not match
        valid = false; }                        // then it is an error
    else
      valid = false;          // an error if no saved symbol to match
i++; }                                          // go get next symbol
// SOMETHING TO WORRY ABOUT AFTER THE END OF THE LOOP
return valid; }                                 // return the error status
```

还需要考虑一个问题。如果在循环结尾将标志valid设置为false，这个值应该返回给调用者，这是毫无疑问的——因为输入表达式是不合法的。但是，如果标志仍然为true，程序也不应该立刻就告诉客户程序表达式是正确的。首先，还应该检查数组store[]中是否还有多余的符号，没有被表达式中的右括号匹配。如果是这种情形（idx>0），表达式也是不合法的，应该把valid设置为false。

checkParen()函数及其相关的测试代码如程序8-10所示。整个程序中有大量的if语句，这意味着该函数必须被调用多次才能证明其正确性。所以，我们在测试用例和函数之间编写了一个辅助函数checkParenTest()，此函数调用checkParen()，并打印输入表达式和函数执行的结果。图8-10显示了程序的运行结果。

程序8-10 直接访问底层数据表示的例子

```
#include <iostream>                             // No encapsulation yet
```

```
#include <cstring>
using namespace std;

char buffer[81];  char store[81];              // global data

bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;               // initialize data
  while (buffer[i] != '\0' && valid)          // end of data or error?
  { c = buffer[i];                            // get next symbol
    if (c=='(' || c=='[')                     // is next symbol left?
      { store[idx] = c;  idx++; }             // then save it away
    else if (c==')' || c==']')                // is next symbol right?
      if (idx > 0)                            // does saved symbol exist?
      { idx--;  sym = store[idx];             // get the last symbol
        if (!((sym=='(' && c==')') ||
              (sym=='[' && c==']')))          // if they do not match
              valid = false; }                // then it is an error
      else
        valid = false;                        // error if no symbol to match
    i++; } // go get next symbol
    if (idx > 0) valid = false;               // unmatched left symbols: an error
    return valid; }                           // return the error status

void checkParenTest(char expression[])        // test harness
{ strcpy(buffer,expression);
  cout << "Expression " << buffer << endl;    // print the expression
  if (checkParen())                           // validate it
    cout << "is valid\n";                     // print the result
  else
    cout << "is not valid\n";
}
int main()                                    // test driver
{ checkParenTest("a=(x[i]+5)*y;");            // first test run: valid
  checkParenTest("a=(x[i)+5]*y;");            // second test run: invalid
  return 0;
  }
```

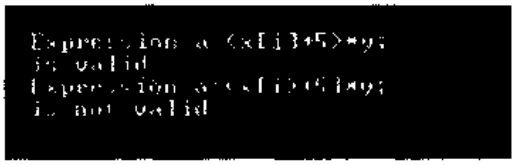


图8-10　程序8-10的输出结果

与本章前面的例子相似，我们将尝试封装checkParen()中的代码，将它从符号表示中抽取出来，这样检查符号的算法就不再依赖于具体的符号。例如，如果代码要处理花括号，则算法也应该对花括号进行判断。但是，如果应用程序处理的表达式中包含花括号（或者其他的配对符号），就必须修改函数checkParen()，该函数甚至可能需要一个不同的函数名，因为它将检查多种括号。

服务器函数应该对客户代码诸如左、右符号的具体细节和符号匹配的规则进行封装处理。例如，这里有三个访问函数可以完成此工作。我们将字符数组buffer[]的下标参数传递给函数isLeft()和isRight()，这两个函数根据此下标所指示的字符返回真或者假。对于

函数symbolsMatch()，我们将传送分别指示数组buffer[]和数组store[]的两个下标，然后函数判断这两个下标所指示的符号是否匹配，并相应地返回true或者false。

```
bool isLeft (int i)
{ char c = buffer[i];                          // get symbol from buffer
  return (c=='(' || c=='['); }                 // check if it is a left symbol

bool isRight (int i)
{ char c = buffer[i];                          // get symbol from buffer
  return (c==')' || c==']'); }                 // check if it is a right symbol

bool symbolsMatch (int idx, int i)
{ char sym = store[idx], c = buffer[i];        // get two symbols to match
  return (sym=='('&&c==')')||(sym=='['&&c==']');}  // do they match?
```

程序8-11显示了使用这些访问函数的代码。如果应用程序要处理花括号，需要修改的是访问函数isLeft()、isRight()、symbolsMatch()，而checkParen()或者其他客户代码都无需修改。程序8-11的输出结果和程序8-10一样。

程序8-11 具有共享信息的封装例子

```
#include <iostream>                            // Bad distribution of knowledge
#include <cstring>
using namespace std;

char buffer[81];  char store[81];

bool isLeft (int i)
{ char c = buffer[i];                          // get symbol from buffer
  return (c=='(' || c=='['); }                 // check if it is a left symbol

bool isRight (int i)
{ char c = buffer[i];                          // get symbol from buffer
  return (c==')' || c==']'); }                 // check if it is a right symbol

bool symbolsMatch (int idx, int i)
{ char sym = store[idx], c = buffer[i];        // get two symbols to match
  return (sym=='('&&c==')')||(sym=='['&&c==']');}      // do they match?

bool checkParen ()
{ char c; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;                // initialize data
  while (buffer[i] != '\0' && valid)           // end of data or error?
  { c = buffer[i];                             // get next symbol
    if (isLeft(i))                             // is next symbol left?
      { store[idx] = c;  idx++; }              // then save it away
    else if (isRight(i))                       // is next symbol right?
      if (idx > 0)                             // does saved symbol exist?
      { idx--;                                 // get the last symbol
        if (!symbolsMatch(idx,i))              // if they do not match
              valid = false; }                 // then it is an error
      else
        valid = false;                         // error if no saved symbol to match
  i++; }                                       // go get next symbol
  if (idx > 0) valid = false;                  // unmatched left symbols: an error
  return valid; }                              // return the error status
```

```
void checkParenTest(char expression[])     // test harness
{ strcpy(buffer,expression);
  cout << "Expression " << buffer << endl;  // print the expression
  if (checkParen())                         // validate it
    cout << "is valid\n";                   // print the result
  else
    cout << "is not valid\n"; }
int main()                                  // test driver
{
  checkParenTest("a=(x[i]+5)*y;");          // first test run: valid
  checkParenTest("a=(x[i)+5]*y;");          // second test run: invalid
  return 0;
  }
```

这样的数据封装好吗？不是很好。符号表示对于客户代码来说的确是隐藏的，但是服务器函数需要知道更多的符号表示和匹配规则。它还需要和客户代码共享数组buffer[]和数组store[]的信息。客户和服务器的关注域还是没有分开。共享这些数组的信息没有必要，只要客户代码知道就可以了。和其他共享信息的情况一样，当设计改变时，客户和服务器的函数都需要改变。如果修改了这些数组的名字或者将数组改成链表的表示方法，受影响的不止是函数checkParen()，在本设计中，符号的访问函数也需要修改。如果我们觉得使用全局数组不合适，甚至连符号访问函数的界面都需要修改——因为要将这些全局数组作为参数传递给访问函数。

这种破坏信息隐藏的形式比较少见。通常都是客户代码需要了解多余的信息，但本例显示出服务器代码也可能了解了多余的信息。服务器函数应该只知道一个数据结构，并对其他程序部分隐藏数据结构相关的信息。

为了保证我们编写的C++代码的质量，我们应该经常考虑共享信息的问题。在本书中会不断提醒大家这一点。

为了消除共享信息的缺点，我们再次设计程序，改变职责的分配方式。我们通过传递符号本身而不是下标给服务器函数，从而对服务器函数隐藏数组下标。在程序8-12所示的版本中，符号访问函数isLeft()、isRight()、symbolsMatch()只需要知道有关符号，而不必知道客户代码存储这些符号的方式。只有客户代码才知道数组。

程序8-12　一个更好的封装例子

```
#include <iostream>                          // Better distribution of knowledge
#include <cstring>
using namespace std;

bool isLeft (char c)
{ return (c=='(' || c=='['); }               // check if it is a left symbol

bool isRight (char c)
{ return (c==')' || c==']'); }               // check if it is a left symbol

bool symbolsMatch (char c, char sym)
{ return (sym=='('&&c==')')||(sym=='['&&c==']');}     // do they match?

bool checkParen (char buffer[])              // expression in parameter
{ char store[81];                            // local array
  char c,sym; int i, idx; bool valid;
```

```
    i = 0; idx = 0; valid = true;                    // initialize data
    while (buffer[i] != '\0' && valid)               // end of data or error?
    { c = buffer[i];                                 // get next symbol
      if (isLeft(c))                                 // is next symbol left?
        { store[idx] = c;  idx++; }                  // then save it away
      else if (isRight(c))                           // is next symbol right?
        if (idx > 0)                                 // does saved symbol exist?
          { sym = store[--idx];                      // get the last symbol
            if (!symbolsMatch(c,sym))                // if they do not match
                  valid = false; }                   // then it is an error
      else
        valid = false;                               // error if no saved symbol to match
    i++; }                                           // go get next symbol
    if (idx > 0) valid = false;                      // unmatched left symbols: an error
    return valid; }                                  // return error status

  void checkParenTest(char expression[])
  { cout << "Expression " << expression << endl;     // print expression
    if (checkParen(expression))                      // validate it
            cout << "is valid\n";                    // print the result
    else
            cout << "is not valid\n"; }

  int main()
  {
    checkParenTest("a=(x[i]+5)*y;");                 // first test run: valid
    checkParenTest("a=(x[i)+5]*y;");                 // second test run: invalid;
    checkParenTest("a=(x[i]+5]*y;");                 // third test run: invalid;
    return 0;
    }
```

在程序的这个版本中，数据封装得更好了，关注域的独立性也更加合理。客户代码只需要知道数组和下标，服务器函数只需要知道符号和匹配规则。

在客户函数中，数组buffer[]的使用是很自然的：它是checkParen()要处理的数组，因此封装这个数组没有多大意义。如果按步骤执行表达式处理，那么函数checkParen()就是进行表达式校验和评估的表达式访问函数。

但是，checkParen()使用了另一个数组store[]，对这个数组的操作增加了客户代码的复杂性。程序员必须决定是否将下标idx初始化为0、1或者别的什么数。当符号保存在数组store[]中时，程序员必须决定先在数组中保存符号再把下标加1，还是先把下标加1再保存符号。当从数组中读取符号时，程序员又必须决定先读取符号再把下标减1，还是先把下标减1再读取符号。（注意，对后两个问题的不同回答效果是不同的。）而且，当checkParen()函数检查数组store[]中是否还有尚未匹配的符号时，程序员必须决定应该将下标与0、1还是其他值相比较。

要回答这些问题并不难，因为我们的程序的确很小。但和其他相似的问题综合在一起时，就会累积复杂性并增加开发阶段特别是维护阶段的出错率。更重要的是，这些问题和函数checkParen()实现的算法并没有任何关系——该函数只需要扫描符号，保存左符号，然后在找到右符号时再获取它们就可以了。每个函数都应该只处理一个未被封装的数据结构，而对于checkParen()函数来说，这个数据结构应该是数组buffer[]，而不是store[]。

因此，这个例子设计的下一步就是在单独的数据结构中封装数组store[]及其下标idx，然后提供访问函数，以供checkParen()使用来访问数据结构的元素。

```
struct Store {
  char a[81];                    // array for temporary storage
  int idx; } ;                   // index to first available slot

void initStore (Store &s)
  { s.idx = 0; }                 // initialize the empty store

bool isEmpty (const Store& s)
  { return (s.idx == 0); }       // check whether the store is empty
void saveSymbol (Store &s, char x)
{ s.a[s.idx] = x;                // save the symbol in the store
  s.idx++; }

char getLast(Store &s)
{  s.idx--;                      // get back the last symbol saved
  return s.a[s.idx]; }
```

有经验的程序员一看到上面的代码，就可能知道这是一个用固定长度数组实现的普通栈。如果不熟悉这个数据结构也没关系，这并不重要。重要的是将客户代码与数据表示的所有细节分开的访问函数，访问函数让客户代码以函数调用的形式描述自己的算法。（见程序8-13）

程序8-13　封装临时存储的store[]

```
#include <iostream>                          // Encapsulation with info hiding
#include <cstring>
using namespace std;

struct Store {
  char a[81];                                // array for temporary storage
  int idx; } ;                               // index to first available slot

void initStore (Store &s)
  { s.idx = 0; }                             // initialize the empty store

bool isEmpty (const Store& s)
  { return (s.idx == 0); }                   // check whether the store is empty

void saveSymbol (Store &s, char x)
{ s.a[s.idx++] = x; }                        // save the symbol in the store

char getLast(Store &s)
{ return s.a[--s.idx]; }                     // get back the last symbol saved

bool isLeft (char c)
{ return (c=='(' || c=='['); }               // check if it is a left symbol

bool isRight (char c)
{ return (c==')' || c==']'); }               // check if it is a left symbol

bool symbolsMatch (char c, char sym)
{ return (sym=='('&&c==')')||(sym=='['&&c==']');}     // do they match?

bool checkParen (char buffer[])              // expression in parameter
{ Store store;                               // array is encapsulated
```

```
  char c,sym; int i; bool valid;
  i = 0; initStore(store); valid = true;   // initialize data
  while (buffer[i] != '\0' && valid)       // end of data or error?
  { c = buffer[i];                         // get next symbol
    if (isLeft(c))                         // is next symbol left?
        { saveSymbol(store,c); }           // then save it away
    else if (isRight(c))                   // is next symbol right?
      if (!isEmpty(store))                 // does saved symbol exist?
      { sym = getLast(store);              // get the last symbol
        if (!symbolsMatch(c,sym))          // if they do not match
              valid = false; }             // then it is an error
       else
         valid = false;                    // error if no saved symbol to match
    i++; } // go get next symbol
    if (store.idx>0) valid=false;          // error: unmatched left symbols
    return valid; }                        // return the error status

void checkParenTest(char expression[])
{ cout << "Expression " << expression << endl;     // print expression
  if (checkParen(expression))              // validate it
          cout << "is valid\n";            // print the result
  else
          cout << "is not valid\n"; }

int main()
{ cout << endl << endl;
  checkParenTest("a=(x[i]+5)*y;");         // first test run: valid
  checkParenTest("a=(x[i)+5]*y;");         // second test run: invalid;
  checkParenTest("a=(x{i]+5]*y;");         // third test run: invalid;
  cout << endl << endl;
  return 0;
  }
```

在数次改进代码的过程中，我们都力图使程序的注释行保持一致。现在我们回到程序8-10（第一个版本），把它和程序8-13比较。我们会发现对于前一个未封装的程序来说，行注释是非常有用的，它们解释了操纵数据表示的意义。但对于后一个使用封装的版本来说，这些注释就显得无关紧要了，它们仅仅是重复其代码所做的工作，而代码的意义已经由所调用的服务器函数的函数名清楚地表达了。

在这个版本的代码中，再没有任何数据表示的细节干扰数据操纵的意义，开发人员和维护人员的注意力也不会分散，而是划分成有限的三个区域。客户代码和服务器代码之间也不再共享数据结构的信息，访问数据的职责推向服务器函数。

8.6 用函数实现封装的不足

以上介绍的是编写软件的很好方法。但是，如果仅仅使用函数实现数据封装和信息隐藏会有一些不足。C++试图通过引入类来消除这些缺点。

缺点之一是这种访问函数没有明确地向维护人员表明设计人员的思路，即这些访问函数是属于一起的，它们访问的是相同的数据结构。在本章的例子中，我们把相关的服务器函数放在一起，让读者很容易明白。一个更好的方法是把函数isLeft()、isRight()和symbolsMatch()放到一个文件里（这些函数都是访问符号的），而把函数initStore()、isEmpty()、saveSymbol()和getLast()放到另一个文件中（这些函数都是访

问临时存储数组的）。

在实际的编程工作中，访问一个数据结构的函数常常和访问另一个数据结构的函数混杂在一起，它们可能按字母顺序排列，这样数据结构及其访问函数之间的关系就变得很不明显。即使把这些相关的函数放在没有外来函数的单独文件中，这种解决方案也是人为的，并不是C++语言支持的。在C语言（和其他一些早期语言）中，没有任何语言机制可以明确指出一些函数逻辑上是属于一起的。C++提供了完美的解决方法——即在类界限内（开花括号和闭花括号之间）将数据及其相关访问函数绑定在一起。正是这个类的界限指出了数据和函数是属于一起的，这些属于一起的函数就不会分散在其他无关的函数之间。

用访问函数实现封装的第二个缺点是这种封装是自发的。客户端代码程序员（或者维护人员）可以使用访问函数，也可以忽略访问函数而直接访问数据结构的域。语言规则并不会防止后一种做法。例如，程序8-13的函数checkParen()的结尾处，我们检查store[]中是否还有符号在checkParen()的调用过程中没有匹配。正确的做法应该是使用访问函数isEmpty()：

```
if (!isEmpty(store)) valid=false;  // error: unmatched left symbols
```

相反，我们走了捷径，直接使用了程序员定义类型Store的结构域名idx。

```
if (store.idx>0) valid=false;      // error: unmatched left symbols
```

这样，使用封装得到的好处就大打折扣了。客户代码的意义不再是自解释性的，而是需要从注释和上下文中推导出来。维护人员需要处理数据访问和数据操纵混杂在一起的代码，因此其工作复杂化了。如果数据域idx的名字需要改变，例如改成top（这个名字更好且更加常用），客户代码也需要进行相应的修改。客户代码和服务器代码之间的依赖性就使得代码更加复杂。因此，依赖程序员的好心来提供封装并不太好，C++提供的解决方法是允许代码的设计人员使用private访问修饰符，这样就不可能破坏封装性。

使用访问函数实现封装的第三个缺点是这种函数是全局函数。它们的名字是全局名字空间的一部分，因此可能和其他函数名相冲突。这样，处理程序不同部分的程序员就不得不协调合作以避免命名冲突，这就迫使程序员了解程序的其他部分，而这本来是不必要的。

C++通过提供块域、函数域、文件域和程序域以外再引入类作用域来解决这个问题。每个作为类成员（数据成员或者成员函数）定义的名字都是在类作用域中定义的。这就消除了名字冲突。程序员不必知道程序其他部分使用的名字，除非他们要使用这些名字。这样就降低了程序员之间协调合作的工作量。

还有一个缺点就是客户代码必须显式地初始化许多数据结构。例如程序8-13中的store变量就是通过显式调用函数initStore()来初始化的。这又扩展了客户维护人员的关注域，也可能导致使用还没有被恰当初始化的数据。

C++通过特殊的构造函数解决了这个问题，构造函数将职责从客户代码推向服务器代码。每当类的对象被创建时，这些构造函数将会被隐式地调用。在该函数中，服务器类的设计人员指定类对象是如何被初始化的。通过在客户端代码程序员和服务器程序员之间划分职责，就让服务器程序员负责处理初始化工作，客户代码的设计人员就从这个职责中解脱出来。C++还提供另一类型的特殊函数——析构函数，这些函数在类对象被撤销时隐式地调用，它们还回动态内存和对象可能需要的其他资源，也减轻了客户端代码程序员还回资源的重任。

除此之外，C++的类还提供了大量的其他方法，提高数据和操作的绑定、服务器数据域名

的封装，更有效地对客户代码隐藏服务器设计细节，将职责从客户推向服务器，并且避免客户和服务器代码之间的依赖性。

C++的类可以极大地提高软件质量，我们将在下面的章节中详细讨论它们。

8.7 小结

在这一章，我们学习了如何使用C++中函数作为程序设计的主要工具。对于给定的程序功能，用C++代码可以有很多种实现该功能的方法。

在函数之间分配工作的目的是，程序的函数可以分开来理解和维护，并且易于在其他的上下文中重用。如果需要客户的设计人员（或维护人员）在多处地方阅读代码以理解和修改程序，这样的设计就会阻碍代码重用和维护。

可读性和组件独立性的标准太过概括和抽象，应该对经验不足的程序员提供更加具体的技术标准。在本章，我们讨论了与内聚性、耦合度相关的传统标准和表现为数据封装、信息隐藏的面向对象标准。我们也讨论了新的标准，例如将职责从客户函数推向服务器函数，避免将应该属于同一代码段的功能分开，分开关注域并限制组件间的共享信息，以及在代码中而不是注释中将开发人员的意图传达给维护人员。

内聚性描述了函数中各个部分之间的逻辑相关程度。内聚性程度高的函数只对一个对象做一件事情；内聚性程度低的函数可能完成多个计算任务。为了提高内聚性，我们应该进行重新设计：将不应属于一起的操作分开在不同的函数中，而不是放在相同的函数中。内聚性不是一个很关键的标准，它应该作为其他标准的补充性考虑因素。

耦合度描述了服务器函数及其客户函数之间的界面。松耦合度说明两者的独立性高。最强的耦合度形式是使用全局变量，这要求写客户和服务器函数的开发人员协调合作。当函数移植到其他上下文中重用时，也要使用相同名字的全局变量。为了分析这些函数之间的数据流，我们需要研究客户和服务器函数的整个代码。

通过参数进行通信的函数比较容易重用。开发人员只需对参数的个数和类型进行协调，而不需要考虑参数名。只研究函数接口而不用研究整个代码就可以理解数据流。为了更好地获得使用参数传递的好处，我们应该应用本章和前一章描述的参数传递使用指南。

为了降低代码的耦合度，我们应该重新为函数分配任务，以让在不同函数中执行的操作移到相同的函数中。这就消除了在函数间通信的必要性。开发人员应该随时考虑哪些通信是必要的和哪些通信是可以避免的。这是程序员进行程序设计的一个重要工具。

数据封装是一种程序设计方法，让客户代码从它所需要的数据域名中脱离出来，转为由访问函数代替客户函数访问这些域。客户函数的代码就可以表示成为对服务器函数的调用，而不是表示成数据域。使用封装可以提高可维护性，因为它在程序中创建了两个独立的问题域。当数据设计改变时，修改访问函数而保持客户函数不变；当程序应用功能改变时，修改客户函数而保持访问函数不变。如果不使用封装，我们就不得不检查代码中的每一个部分以进行可能的修改。

信息隐藏也是一种程序设计方法，它进一步将客户函数从数据表示中隔离开来。要选择那些能够代表客户函数完成工作的访问函数。客户代码表示成对这些服务器函数的调用，这些服务器函数的函数名就描述了客户代码的算法。这种方法进一步提高了程序的可维护性和重用性。

如果我们合理地使用了上述编程思想，那么客户代码肯定会变得更加面向对象，因为它表示成对数据结构的操作。然而，用函数来实现面向对象编程还是有一些问题没法解决。没有语言级的机制指出数据和访问函数是属于一起的；维护人员必须通过查看代码推导出这些关系。而且，访问函数的函数名对程序作用域而言是全局的，可能产生名字冲突问题。此外，封装是自发的，并且基于程序员的原则。如果客户函数的开发人员在客户函数中使用了数据域的名字，封装得到的好处就消失了。

为了解决这些问题，C++提供了类结构。类的边界表明了数据和函数是属于一起的。每个类都有自己单独的作用域，有相同函数名的不同类中的访问函数也不会彼此冲突。类的开发人员可以声明数据（和函数）是私有的，以防止客户函数直接访问它。

类是令人振奋的技术！它为编写高质量的代码开辟了新的空间。从下一章开始我们将会集中论述C++的类。

第9章 作为模块单元的C++类

在前一章，我们总结了使用函数作为程序构造块进行面向对象程序设计的基本原理。按照面向对象的编程思想，客户代码调用服务器的访问函数，而不是直接访问和修改数据域。服务器函数提供的操作就是为了实现客户代码的目标。各个函数分担职责，因此客户函数不知道数据表示，服务器函数不了解客户代码的算法。

这就建立了不同关注域的独立性。当修改访问函数时，维护人员没有必要在客户函数中进行相应的修改（如果服务器界面没有变化）。改变客户函数时，维护人员也不必考虑服务器函数中的数据处理细节，它们也不需要修改。客户代码由对服务器函数的调用组成，而不是具体的数据操纵。我们将应该属于一起的部分放在一起（而不是分开来），让函数之间互相独立，从而进一步增加可维护性和重用性。在前一章，我们所给出的对象图显示了服务器函数之间逻辑相关，以及与它们所访问的数据也逻辑相关。

可以看出，使用函数来实现面向对象的编程思想，其代码编排完全由程序员决定。服务器函数可放在源代码的不相关位置，这样，维护人员就可能不会注意到它们彼此相关及和某些数据表示相关。客户函数也无法使用封装，而是可以直接访问数据表示，并在程序的不同部分之间建立连接和依赖关系。

由于时间紧迫或者因为人的注意力的局限，程序员可能在函数之间引入依赖性。互相依赖的函数很难开发，因为开发不同但互相相关的函数的程序员必须互相协调，但令人惋惜的是，这种协调统一常常会糟糕地被打破。

互相依赖的函数之所以难于维护，是因为维护人员必须在修改代码前去研究这些依赖性。这样的函数更难重用，因为不能单独移到另一个程序中，它们还需要相关的数据和其他函数。因此，程序员需要能够掌握的程序设计语言帮助他们弥补这些不足。为了帮助程序员编写更好的代码，C++提供了绝妙的语言构造单元——类，它可以将数据表示和对数据的操作（函数）在物理上捆绑在一起，而不只是在设计人员的概念上捆绑在一起。将数据和操作绑定在一起就支持了数据封装和信息隐藏的概念。

本章，我们将仔细探讨C++的类。我们将学习类的语法和语义，也将学习如何定义类成员，包括数据成员和成员函数。本章还将解释如何指定对类成员的访问权限；如何在一个或者多个程序中实现类；如何定义对象（类实例）；以及如何操纵对象，即如何发送消息，如何将对象作为参数传递，如何从函数返回对象等。

我们还会讨论特殊的成员函数，如构造函数和析构函数，这些函数使用不正确往往会造成误解。我们也会进一步讨论第7章中讨论过的const修饰符，以帮助开发人员将他们在进行设计时的想法思路传达给维护阶段的维护人员。另一种特别的数据成员和成员函数是静态数据成员和静态数据函数。静态成员和函数帮助设计人员描述对类的所有对象都通用的类特征。

这确实是充满挑战的工作。在本章结束时，我们应该能够熟练地使用较大的模块化单元（类）而不是较小的模块化单元（函数）。当然，我们也可能会因为必须掌握大量的技术细节而觉得沮丧，这是很自然的。C++是一种大而复杂的语言，必须花费不少的时间来熟悉它的

概念、实用细节和陷阱。如果有人夸下海口说学C++很容易，他或者是撒谎或者是还没有意识到其艰巨性和复杂性。如果觉得受挫和困惑，不要试图一下子学习所有的知识，要循序渐进。可以先跳过本章中的某些部分阅读下一章，然后再回过头来反复研读。修改例子的代码，用C++代码以不同的方法实现它，就会发现C++程序设计的不同元素之间有非常优美的内部逻辑联系，根本不难使用。当然，要到达这种“境界”需要大量的练习。

回过头来掌握使用类的C++基础知识是很重要的。很多程序员跳过这个阶段，急于求成地去学习类似继承和多态这样复杂的论题，而没有坚实的基础。结果变得越来越糊涂，写的程序难以理解、维护和重用。C++提供了许多工具，但是这些工具可能会被误用（就像枪、汽车或计算机一样）。使用好的工具不一定能够保证好的结果。要想有效地运用这些工具还在于程序员自己的努力。

9.1　基本的类语法

在C++中引入类的目的是支持面向对象程序设计，并消除使用较小的模块化单元函数所导致的缺点。

类构造的首要目标是将数据和操作捆绑在一个语法单元中，以表示这些代码元素是属于一起的。另一个主要目标是消除名字冲突，以便不同类的数据和函数可以使用相同的名字而不会有冲突；第三个重要的目标是允许服务器设计人员控制从外部（客户代码）对类元素的访问。第四个目标是支持数据封装、信息隐藏、将责任从客户代码推到服务器代码、创建单独的问题域、减少共享信息的交换量、减少负责程序不同部分的程序员之间的协调。

这些目标都是使用函数进行面向对象程序设计实践的自然扩展。如果将C++的类看做是另外一个语法结构而不将它与上述的四个目标联系起来，使用类并不会提高代码的质量。一定要特别注意这四个目标，每次向程序添加类时都应该努力实现这四个目标。

类是C++和面向对象程序设计的核心。类是程序员创建新的数据类型的工具，它比函数式程序更加贴近客观现实世界的对象的行为。有些程序员认为继承和多态才是面向对象程序设计的核心。但是，如果正确地使用类并达到上述四个目标，每个大型的C++程序都得益于C++类的使用。一个设计正确的C++程序由组件（模块）组成，这些组件互相合作以完成共同的任务，但又彼此独立以便分开维护。

9.1.1　绑定数据与操作

结构类型也通过合并数据域支持绑定的概念。它允许我们将不同的组件合并在一个复合数据对象中。这些复合数据对象可以作为一个整体处理，例如，作为参数传递给函数；也可以提供对其元素的单独访问。

但是一个结构类型的定义只是对数据集合而不是它们的行为进行建模。服务器程序员需要提供工具操纵这些数据，即提供一系列的访问函数，以代表客户函数访问和操纵这些数据。在“函数式”和“过程式”程序设计中，数据和算法在语法上是分开的，它们只是在程序员的思想中而不是在代码中互相相关。在讨论第8章的一个例子时，我们画出了对象图，以表示函数和数据在逻辑上是属于一起的。

当一个程序由函数组成时，不同函数间的关联性不明显，每个函数都可以访问程序中的每个数据。这使得程序开发、特别是维护和重用更加困难。

体现数据和使用数据的代码相关的惟一方法是，将数据项和相关函数原型放到同一个头文件中或者一个被分开编译的源文件中。但一个磁盘文件是硬件（或者操作系统）概念，而不是编程语言的概念。因此，C++扩展了struct类型的功能，将包含值的数据成员和对这些值进行操作的成员函数绑定在一起。

这样产生的对象代表了比较大的模块化单元。客户的程序员将注意力放在数据和相关的函数上，而不是其联系并不明显的独立的函数上。

在设计良好的C++程序中，类的数据只能通过属于同一类的成员函数来访问。客户代码被表示成为操作的集合而不是对数据的访问。这就降低了客户设计人员和维护人员的工作量。

形式上来说，如果将域放在一个struct定义中，实际上就是创建了一个C++类。

```
struct Cylinder {                    // programmer-defined type (class)
  double radius, height;  } ;        // end of class scope
```

在C++中，关键字struct和class（几乎）是等效的。对刚才定义的类Cylinder，我们可以定义这个类的对象（或实例或变量），可以设置对象域的值。还可以把对象视作一个单个的实体（例如，将它作为函数参数传递，或者存储在一个磁盘文件内），也可以在计算中使用它的单个部分。

在下面的例子中，main()函数定义了两个Cylinder类型对象（变量和实例），并初始化它们，然后比较这两个圆柱体的体积。如果第一个Cylinder对象的体积比第二个对象的体积小，就把第一个Cylinder对象的体积放大20%，再输出第一个Cylinder对象新的半径和高。这和我们在第8章开始时讨论的例子相似。

```
int main()
{ Cylinder c1, c2;                                          // program data
  c1.radius = 10; c1.height = 30; c2.radius = 20; c2.height = 30;
  cout << "\nInitial size of first cylinder\n";
  cout <<"radius: " <<c1.radius <<" height: " <<c1.height <<endl;
  if (c1.height*c1.radius*c1.radius*3.141593                // compare volumes
    < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2;  c1.height *= 1.2;                    // scale it up and
    cout << "\nFirst cylinder changed size\n";              // print new size
    cout <<"radius: " <<c1.radius <<" height: " <<c1.height <<endl; }
  else                                                      // otherwise do nothing
    cout << "\nNo change in first cylinder size" << endl;
  return 0; }
```

这段代码显式使用了数据域的名字。客户代码访问了域的值，完成任何必需的工作（包括计算体积、缩放圆柱体和输出）。对Cylinder设计的修改不仅会影响Cylinder结构，还会影响客户代码。维护人员也必须跟踪计算的每一步以了解操作的意义（包括计算体积、缩放圆柱体和输出）。为了确保对象的所有维数据都被初始化、放大和打印，我们还需要参考类Cylinder的定义。为同一个项目或者其他项目重用这些操作（包括计算体积、缩放圆柱体和输出）也是很困难的，因为它们都依附在客户代码的上下文中。

使用访问函数，将对结构域的操作封装起来，就可以消除这些缺点。这些函数如：setCylinder()、printCylinder()、getVolume()、scaleCylinder()。程序9-1显示了这种版本的客户代码和服务器代码，程序的运行结果如图9-1所示。

程序9-1 为了客户代码使用访问函数的例子

```
#include <iostream>
```

```
using namespace std;

struct Cylinder {                                        // data structure to access
  double radius, height; } ;

void setCylinder(Cylinder& c, double r, double h)
{ c.radius = r;  c.height = h; }

double getVolume(const Cylinder& c)                      // compute volume
{ return c.height * c.radius * c.radius * 3.141593; }

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor;  c.height *= factor; }             // scale dimensions

void printCylinder(const Cylinder &c)                    // print object state
{ cout << "radius: " <<c.radius << " height: " <<c.height <<endl; }

int main()                              // pushing responsibility to server functions
{ Cylinder c1, c2;                                       // program data
  setCylinder(c1,10,30); setCylinder(c2,20,30);          // set cylinders
  cout << "\nInitial size of first cylinder\n";
  printCylinder(c1);
  if (getVolume(c1) < getVolume(c2))                     // compare volumes
  { scaleCylinder(c1,1.2);                               // scale it up and
    cout << "\nFirst cylinder changed size\n";           // print new size
    printCylinder(c1); }
  else // otherwise do nothing
    cout << "\nNo change in first cylinder size" << endl;
  return 0;
  }
```



图9-1　程序9-1的输出结果

这个例子和程序8-7类似，而且我们到此为止所演示的内容也没有超出一个普通的结构类型的功能。下面我们更进一步，把数据域和函数结合在相同的类中。在下面的例子中，用开花括号、闭花括号以及结尾的分号表示类Cylinder的语法边界。

这个类有两个域，或称之为数据成员：radius和height。除了数据成员之外，还包含了4个成员函数（成员函数的另一个术语是方法，这种说法来自Smalltalk和人工智能）。成员函数和全局函数有相同的语法：它们可以有参数和返回值，每个函数的内部变量都有其作用域。与全局函数不同的是，成员函数在类的边界（花括号）内定义。现在我们可以清晰地看出setCylinder()、printCylinder()、getVolume()、scaleCylinder()还有数据域radius、height是属于一起的。

```
struct Cylinder {                                        // start of class scope
  double radius, height;                                 // class data members
void setCylinder(double r, double h)                     // class member functions
{ radius = r;  height = h; }                             // set field values
```

```
double getVolume()
{ return height * radius * radius * 3.141593; }        // compute volume
void scaleCylinder(double factor)
{ radius *= factor;  height *= factor; }               // scale dimensions
void printCylinder()                                   // print object state
{ cout << "radius: " <<radius << " height: " <<height <<endl; }
} ;                                                    // end of class scope
```

增加成员函数并不会改变结构的基本属性，它是用来定义这个类的对象的模板。

```
Cylinder c1, c2;      // space for two object instances is allocated
```

在讨论面向对象设计和编程时，我们很自然会用到术语“对象”。不幸的是，这个术语不止一个含义。有些人用“对象”来表示对应用程序来说很重要的抽象概念，例如顾客、账号或者事务对象。有些人用这个术语来标识个别的对象，例如术语某个特定顾客的账号。还有些人在使用这个术语时根本不知道是什么意思，而是希望其他人能够猜测出来。这里不立即做出选择，但是我们最好不要成为第三类人。

在本书中，我们将术语变量、实例、类实例、类对象和对象实例作为同义词使用。它们都表示在程序执行某一阶段被分配内存空间（在栈或堆中）的程序实体；它们属于某一特定的存储类别，遵循作用域规则。我们会避免使用“对象”这个术语，如果偶尔用到了，就表示一个程序变量。在大多数情况下，它表示一个程序员定义类型的变量，但是用术语“对象”来描述基本类型的变量也是可以的。这就是该术语的程序设计意义。

在面向对象分析和设计中，术语“对象”常常用来表示有相同属性的一系列潜在实例。这种用法和类（程序员定义类型）概念的接近程度要比和对象实例概念的接近程度更近一些。我们不在这里争论哪一种用法正确。但是，正是因为这种不确定性，如果不能描述术语“对象”的意义，最好就不要使用它。

在一本立足于面向对象程序设计的书中表达这样的观点可能有些不合适。我们反对使用术语“对象”，不加区别地使用这个术语只会导致其意义更加模糊。在大家使用该术语时，一定要清楚其意义，是指程序执行过程中存在于计算机内存中的单个实例，还是指这些潜在实例的通用性描述，即一个用来在程序运行过程中创建特定的实例的C++类。

类成员函数的存在增加了结构定义的大小，但并不会增加结构实例在内存中分配空间的大小，因为其大小仍然为各个域的大小之和（可能需要额外的空间以便对齐）。如每一个Cylinder实例被分配的空间足够容纳两个double类型的值。

9.1.2 消除名字冲突

类的开花括号和闭花括号（以及结尾的分号）形成了类的作用域，就像普通的结构为它的域形成了单独的作用域一样。类作用域嵌套在文件作用域之中，和普通结构的作用域类似。不同之处在于类作用域还可以嵌入函数作用域。

非成员函数（如程序9-1中的访问函数）是全局函数，它们的名字在整个程序中应该是惟一的（除非在文件中被声明为静态static类型，但在文件作用域中只有很小部分的函数可以被声明为静态。在第6章和本章后面部分可以看到关于静态函数的更多说明。）一般来说，只有使用类Cylinder的成员才需要了解函数名，因为它要调用该函数。而实践过程中，每个成员都应该了解这些函数名以避免偶尔的名字冲突。这些信息阻塞了程序员之间的交流渠道。

当函数作为类成员函数被实现时（如下面的程序9-2所示），其函数名在类作用域中是局

部的。因此，不能使用函数名调用成员函数（或者，访问数据成员），例如radius或setCylinder()。我们必须指出这个radius属于哪个Cylinder实例，或者指出哪个Cylinder变量调用函数setCylinder()。

```
c1.radius = 10;                    // radius of c1
c2.setCylinder(20,30);             // setCylinder() for c2
```

在这个例子中，Cylinder变量c1的radius被设置为10，Cylinder变量c2被用来调用setCylinder()成员函数。

如果程序中还使用了另外的类类型，如Circle，它的数据成员也有radius，不会造成名字冲突。

```
struct Circle {
  double radius;                   // it can be integer or anything
. . . } ;
```

为了访问类Circle中的域radius，应用程序必须定义Circle对象实例，并使用它们的名字访问radius域。

```
Circle cir1;  cir1.radius = 10;    // no ambiguity: Circle, not Cylinder
```

类括号中的所有类成员（包括数据成员和成员函数）都有相同的类作用域。因此，它们可以互相按名字访问，不需要指定对类名字或者对象名字的引用（作用域运算符）。例如，成员函数setCylinder()设置域（数据成员）radius和height的值。

```
void setCylinder(double r, double h)       // set field values
{ radius = r;  height = h; }
```

究竟radius和height属于谁的呢？它们是某个Cylinder对象（类实例）的域。当一个成员函数（如setCylinder）在客户代码中被调用时，称为向对象发送消息。客户代码（在类的花括号之外）通过显式地使用对象名、成员函数名和在两者之间的点选择运算符标识消息的目标（即成员函数使用的域所在的对象）。

```
Cylinder c1, c2;                        // potential targets of messages
c1.setCylinder(10,30); c2.setCylinder(20,30);     // messages to c1, c2
```

消息应用于这个类的一个对象实例。执行第一个消息时，setCylinder()内部使用的是对象c1的radius和height。执行第二个消息时，setCylinder()内部使用的是对象c2的radius和height。不但当消息执行过程中域的值被修改时是这样，域的值仅仅用于计算时也是这样。

```
if (c1.getVolume() < c2.getVolume())       // compare volumes
  { c1.scaleCylinder(1.2);  . . .          // scale it up
```

第一个消息中，是用变量c1的域计算体积（不管计算什么）；第二个消息中，是用变量c2的域计算体积。在所有场合，我们使用对象名、点选择运算符和消息名（成员函数名）。这种消息语法和访问（或者修改）结构域的语法一样。我们使用对象名、点选择运算符和域名访问一个域。

```
c1.radius = 40.0;  c1.height = 50.0;       // variable c1 is used
```

程序9-2使用了将数据域和成员函数绑定在一起的类Cylinder，包括客户代码和服务器代码。因为其整体功能和程序9-1一样，只是实现形式有所改变，因此输出结果和程序9-1也

一样。

程序9-2 在类固有作用域中数据和函数绑定的例子

```
#include <iostream>
using namespace std;

struct Cylinder {                                    // start of the class scope
   double radius, height;                            // data fields to access

void setCylinder(double r, double h)                 // set cylinder data
{ radius = r;  height = h; }
double getVolume()                                   // compute volume
{ return height * radius * radius * 3.141593; }

void scaleCylinder(double factor)                    // scale dimensions
{ radius *= factor;  height *= factor; }

void printCylinder()                                 // print object state
{ cout << "radius: " <<radius << " height: " <<height <<endl; }
} ;                                                  // end of class scope

int main()                          // pushing responsibility to server functions
{ Cylinder c1, c2;                                   // define program data
  c1.setCylinder(10,30); c2.setCylinder(20,30);      // set cylinders
  cout << "\nInitial size of first cylinder\n";
  c1.printCylinder();
  if (c1.getVolume() < c2.getVolume())               // compare volumes
  { c1.scaleCylinder(1.2);                           // scale it up and
    cout << "\nFirst cylinder changed size\n";       // print new size
    c1.printCylinder(); }
  else                                               // otherwise do nothing
    cout << "\nNo change in first cylinder size" << endl;
  return 0;
  }
```

比较程序9-2和程序9-1，我们很容易看出程序9-1使用单独的全局函数和程序9-2使用与数据捆绑的函数之间的不同。使用单独的访问函数，其数据在函数中被使用的对象变量要作为参数传递给函数。

```
void setCylinder(Cylinder& c, double r, double h)    // access function
{ c.radius = r;  c.height = h; }                     // Cylinder is a parameter
```

合适的对象实例应该在函数调用中当做实际参数使用。

```
setCylinder(c1,10,30); setCylinder(c2,20,30);
```

如果不使用类，没有Cylinder参数就实现setCylinder()函数，或者不将要操作的实际对象传递给函数就进行函数调用都是错误的。

```
void setCylinder(double r, double h)     // nonsense: what Cylinder?
{ c.radius = r;  c.height = h; }

setCylinder(10,30); setCylinder(20,30);  // nonsense: what Cylinder?
```

如果我们将函数设计成类的成员函数，就没有必要把对象作为参数传递了。

```
void setCylinder(double r, double h) // method: no Cylinder parameter!
```

```
{ radius = r;  height = h; }     // data members, not parameter fields!
```

而且，合适的对象实例被指定为函数调用时消息的目标。

```
c1.setCylinder(10,30);              // object c1 as a message target
c2.setCylinder(20,30);              // object c2 as a message target
```

很多初学C++的程序员喜欢同时使用这两种方法。在设计类成员函数时，也将要操作的对象作为参数传递。

```
void setCylinder(Cylinder& c, double r, double h)    // bad method
{ c.radius = r;  c.height = h; }

c1.setCylinder(c1,10,30); c2.setCylinder(c2,20,30);  // bad messages
```

并不清楚为什么有些程序员会写出这样的C++语法，但是，的确经常可以看到这种风格的代码。这样的代码语法正确吗？当然正确，否则程序员不可能使用它。这样的代码语义正确吗？当然正确，否则程序员会用其他的方法。但是，这样的代码实在太不美观了。

注意使用不同的消息目标也可以实现一样的结果。

```
c2.setCylinder(c1,10,30); c1.setCylinder(c2,20,30);     // still bad
```

这样的代码乍一看，还以为将第一个消息传递给变量c2，把第二个语句发送给变量c1，而事实上并非如此。和前面的例子一样，第一个消息仍然设置变量c1的域，第二个消息仍然设置变量c2的域。但是我们必须花费一定的精力才能搞清楚消息的目标和消息本身没有任何关系。

这个糟糕的设计只是一个例子，用来说明用C++很容易写出并不像它看起来那样进行操作的代码，也说明很容易就使用了其实根本和处理无关的程序成分。这种设计常常会增加代码的复杂性，以及客户代码和服务器代码的耦合度。

从写单独的函数转换到写类需要一种风格上的调整。希望大家对两种不同的风格都有所熟悉。

9.1.3 在类之外实现成员函数

注意这些成员函数只有在类作用域即类的花括号中实现才是正确的，如程序9-2所示。如果在类的作用域外实现，应该使用不同的语法。当成员函数和数据成员的关系是通过类声明（规格说明）中的函数原型建立的，而不是像程序9-2通过完整的实现建立的，就使用这样的语法。

```
struct Cylinder {                              // start of class scope
  double radius, height;                       // data fields to access
void setCylinder(double r, double h);          // set Cylinder fields
double getVolume();                            // compute volume
void scaleCylinder(double factor);             // scale dimensions
void printCylinder();                          // print object state
} ;                                            // end of class scope
```

这意味着把函数的定义（函数体）分开来实现。通常，类规格说明被放在扩展名为“.h”的头文件中，函数的实现放在扩展名为“.cpp”或者“.cxx”的源文件中，到底是哪一个文件就要根据具体的编译程序而定。

类声明中的成员函数原型和单独的全局函数原型看起来很相似，惟一的不同是成员函数

原型定义在类作用域内。我们要对类的作用域边界多加注意。程序员很少会忘记写开花括号和闭花括号，但往往会漏掉闭花括号后的分号。但不幸的是，编译程序几乎不会提醒我们漏掉了类规格说明后的分号，而是指出下一行代码有问题。因此，当编译程序提示类规格说明后的代码行有问题时，要检查是否漏掉分号。

警告 程序员有时候会忘记在类的闭花括号后面加上分号。通常编译程序会告诉我们下一行代码出错，而不会指出漏掉分号的那一行代码有错。

当成员函数在类的规格说明之外实现时，我们必须指明成员函数所属的类的名字。这是很自然的，因为每个类都有其作用域，每个类都有可能有成员函数，如getVolume()等。在像Cube、Circle或Account这样的类中使用setCylinder或printCylinder作为成员函数名频率不是很高。对于这些类，程序员更喜欢用setCube()、SetAccount和PrintAccount作为成员函数名。但对于类Cube、Cylinder或Circle，我们也可以使用getVolume()函数，当然，圆的体积可能为零。

像setCylinder()和setAccount()一类的名字在C++出现之前用起来就很有理由，因为C语言不能通过不同的标识（signature）来区分不同的函数。C++就可以区别有相同函数名但不同函数标识的函数（参见第7章7.6节对函数名重载的讨论）。因此，C++中可以用set()函数名代替setCylinder()和setAccount()，因为以Cylinder类型变量为参数的函数set()可以代替setCylinder()，以Account类型变量为参数的函数set()可以代替setAccount()。

C++添加了类作用域，成员函数名冲突就不再是什么重要问题。所以，我们可以用set()代替setCylinder()，print()代替printCylinder()，如此类推。

```
c1.set(10,30); c2.set(20,30);     // Cylinder objects as message targets
```

当编译程序处理一个消息时，它会识别目标对象的名字，并在该对象的定义（或声明）中查找以确定其类型。在这个例子中，编译程序很容易就辨识出对象实例c1和c2是Cylinder类型。接着，编译程序继续查找这个类型的定义（或声明），并判断类型中是否定义了其函数名为消息名的成员函数。如果找到了set()函数，编译程序检查函数接口，即参数的个数和类型。如果参数个数相等但参数类型不匹配，编译程序会寻求可能的类型转换。若类型转换可实现参数类型匹配，编译程序就生成目标代码；否则，编译程序产生错误信息。

编译程序很容易确定目标对象的类型，它只要在源代码中进行查找就可以了（或者，在处理变量声明时已经创建的表中进行查找）。但对于维护人员来说，情况就不一样了。他们必须在代码中搜索，这可能很困难、很耗时而且容易出错。从这个角度来看，一个长函数名可能可以帮助维护人员查找消息的目标定义。

```
c1.setCylinder(10,30);        // objects are Cylinders, right?
c2.setCylinder(20,30);
```

下面，我们再来讨论当类的规格说明只包含成员函数原型时，成员函数的实现。这种实现和在类边界中实现函数完全一样。惟一的不同就是我们必须用类名标注成员函数名。即在类名和成员函数名之间加上作用域运算符。

```
inline void Cylinder::setCylinder(double r, double h)
{ radius = r;  height = h; }                        // set data fields
```

```
inline double Cylinder::getVolume()
{ return height * radius * radius * 3.141593; }   // compute volume

inline void Cylinder::scaleCylinder(double factor)
{ radius *= factor;  height *= factor; }          // scale dimensions

inline void Cylinder::printCylinder()             // print object state
{ cout << "radius: " <<radius << " height: " <<height <<endl; }
```

从我们的角度来看，成员函数setCylinder()的名字实际上不仅是setCylinder()而是类Cylinder的setCylinder()。从语法术语的角度来说，就是用Cylinder::setCylinder()来表示。

分开两部分的类定义（有函数原型的规格说明和分开的类实现）和前面一样定义了相关的类Cylinder。注意，如果在类的规格说明中实现一个成员函数，那么它缺省是内联的；如果分开来实现，缺省就不是内联的，但是可以显式地说明它为内联模式。

我们说过，有函数原型的类规格说明常常放在头文件中，而函数的实现放在单独的源文件中。当然，如果把成员函数的所有或者部分实现放在头文件里也是可以的。在类名被提及的每一个文件中都需要使用include指令引入头文件，例如在客户源文件中、甚至在成员函数被实现的源文件中（因为在类作用域运算符中使用了类名）。

因为连接程序不应该看见多次的函数定义，所有类规格说明应该包含在用于条件编译的预处理指示符内（其他例子可以参见第2章和第5章）。例如，类Cylinder的头文件应该如下所示。

```
#ifndef CYLINDER_H                          // common convention for symbol name
#define CYLINDER_H
#include <iostream>
using namespace std;

struct Cylinder {                           // start of the class scope
  double radius, height;                    // data fields to access
void setCylinder(double r, double h)        // set cylinder data
{ radius = r;  height = h; }
double getVolume()                          // compute volume
{ return height * radius * radius * 3.141593; }
void scaleCylinder(double factor)           // scale dimensions
{ radius *= factor;  height *= factor; }
void printCylinder()                        // print object state
{ cout << "radius: " <<radius << " height: " <<height <<endl; }
} ;                                         // end of class scope
#endif
```

按惯例，我们将这段代码放在文件Cylinder.h中，并将这个符号名用于条件编译CYLINDER_H。在单独的文件中实现成员函数对程序模块化起了很大的帮助。逻辑上来说，成员函数是在类区域中定义的，不论它是否位于类区域，如Cylinder::setCylinder()。正因为如此，这些函数在访问半径和高都为数据成员时不需要使用限定符（作用域运算符）。

把类实现和类界面分开时，必须要使用函数名限定符。如果没有使用限定符，编译程序会以为函数setCylinder()使用全局变量radius和height，而不是类中的数据成员radius和height。

```
inline void setCylinder(double r,double h)     // class scope operator is missing
```

```
{ radius = r;  height = h; }                    // are these data members or what?

inline double Cylinder::getVolume()             // compute volume
{ return height * radius * radius * 3.141593; }

inline void Cylinder::scaleCylinder(double factor)
{ radius *= factor;  height *= factor; }        // scale dimensions

inline void Cylinder::printCylinder()           // print object state
{ cout << "radius: " <<radius << " height: " <<height <<endl; }
```

编译程序把上面的代码视作全局函数的定义，这在C++中是完全合法的。如果在文件作用域中没有声明变量radius和height，编译程序会指示变量radius和height未定义，而不会提醒说漏用了作用域运算符。即编译程序产生了误导的错误信息。大家的第一反应可能是不相信。难道编译程序看不到在类的规格说明中就定义了变量radius和height吗？这一定是编译程序的另一个错误。但是编译程序根本无从知道程序员是因为忘记使用作用域运算符而误定义了全局变量。顺便说明的是，如果在文件作用域中为了其他目的定义了这些名字，编译程序会认为它们指向的是全局变量而不是类数据成员，并不做任何提醒就产生目标代码。

警告 程序员有时候会忘记在成员函数名前添加作用域运算符。编译程序会假设程序员想要实现全局函数，因此会归咎于程序员在函数中使用了未定义的和类数据成员同名的变量。

9.1.4 不同存储方式的类对象的定义

类作用域包括所有的数据成员和成员函数，它嵌套在类声明所在的文件（或另一个类、函数、甚至语句块）中，和其他的函数和/或类一起。只有当类对象在作用域中时才能访问类的成员。

和任何其他类型的变量一样，C++中的类对象（实例、变量）也可以定义为自动、全局、静态或者动态变量。（参见第6章有关于存储方式的讨论）。

对自动变量和全局变量（extern或者static）而言，对象定义时就会隐式地为之分配空间。前面所有定义类实例的例子都是使用自动变量的例子。当程序执行到类实例定义时，就创建了实例。例如，在程序9-2中，当执行到main()中定义c1和c2变量的代码行时，就创建变量c1和c2。

如果将一个对象定义为全局变量，在main()开始执行前就为该对象分配空间。当类实例被定义为静态变量（无论是一个文件的全局变量还是一些函数作用域的局部变量）时，情况也是如此。

这些对象实例的共同点是，必须通过它们的名字引用。为了访问这些对象（及指向它们的引用）的数据域和成员函数，当客户代码需要访问类成员时，客户代码可以使用对象名和点标记。

```
Cylinder x;
x.setCylinder(50,80);
double volume = x.getVolume();  x.radius = 100;
```

对动态变量而言，要使用new运算符显式地为之分配空间。这些对象也不是通过对象名

来定义的，只能通过指向对象的指针来访问。客户函数需要使用指针名（而不是对象名，因为对象实例没有名字）和箭头符号来访问对象的数据成员和成员函数。在下面的例子中，创建了指向Cylinder对象的命名指针，然后通过这个指针来创建并操纵无名的Cylinder类对象。

```
Cylinder* p;                              // no object is created yet
p = new Cylinder;                         // no object name exists
p->setCylinder(50,80);                    // unnamed object is accessed
double volume = p->getVolume();           // same notation
p->radius = 100;
```

如果讨厌使用箭头选择运算符，也可以使用间接引用运算符和点选择运算符。

```
(*p).setCylinder(50,80);            // same as p->setCylinder(50,80);
```

类似地，如果按指针（而不是按值或者按引用）将对象传递给客户函数时，必须使用箭头符号（而不是点符号）。

```
void CopyData(Cylinder *to, const Cylinder &from)       // copy Cylinder
{ to->radius=from.radius;  to->height=from.height; }  // arrow notation

Cylinder x,y;  x.radius=3.0;  x.height=7.0;     // client for CopyData()
CopyData(&y,x);                                 // passing object by pointer
```

自动变量在超出它定义所在的作用域时被撤销。程序员无需进行任何操作以将其内存还回给系统重用。全局变量和静态变量也如此，在超出作用域时被撤销，即在main()函数终止后立刻撤销。不需要进行任何程序处理。

动态变量就不同。我们必须显式地删除它们，因为系统不知道程序员想在什么时候还回动态内存。

```
Cylinder* p = new Cylinder;              // unnamed object is created
p->setCylinder(50,80);                  // unnamed object is accessed
cout << "Volume: " << p->getVolume() << endl;      // arrow operator
delete p;              // unnamed Cylinder is destroyed, pointer is not
```

和任何类型的变量一样，客户访问类实例及其成员都要遵循作用域规则，只有类实例在作用域内时才能被访问。此外，C++允许类的设计人员为程序的其他部分访问类实例建立额外的限制。

9.2 对类成员的控制访问

在前一节里，我们设计了类Cylinder，将它的数据成员和成员函数绑定在一个语法单元中。这个语法解决了用全局函数进行面向对象程序设计时产生的两个问题。

首先，使用全局函数访问数据并不能明显地指示操作和数据是属于一起的。这样就极有可能把本来应该放在一起的函数分离开来，并散布到源代码的不同部分。（这样就让维护人员难以理解代码，也很难修改代码。）其次，全局函数名是全局的。为了避免可能出现的名字冲突问题，即使程序员所开发的程序部分并不一定直接相关，也不得不互相协调。类的语法清晰地表明了数据和函数是属于一起的。类的作用域消除了函数名冲突的可能性。

在本章的开头，我们提到了在C++中引入类功能的另外两个目标：将职责从客户代码推到服务器函数和控制对类成员的访问。

将职责从客户代码推到服务器类是通过正确地选择成员函数来实现的。(有时候，选择数据成员也很重要。)在第8章中，我们讨论了一个例子(见程序8-8)，使用了成员函数setRadius()、getRadius()、setHeight()、getHeight()，强制客户代码完成工作，而不是请求服务器完成工作。相比之下，程序9-2的处理更好一些，它不是通过获得radius和height的值来完成放大、打印或计算体积等工作，而是让客户代码请求类Cylinder的对象来放大、打印自己或者计算自己的体积。

将职责推到服务器是一个重要的概念。它是否有效则是很主观的看法。我们排斥了程序8-8的设计，但是如果这个类用来作为库功能并为大量的用户提供服务，那么这种设计可能是很有用的。对一些用户来说，程序9-2的设计可能过于严格，也许他们还希望计算圆柱体的表面积，非等比例地缩放圆柱体等。但是对另一部分用户来说，程序9-2可能又太过一般化了，他们可能并不需要圆柱体的体积值，而只需知道两个圆柱体对象哪一个大哪一个小(与程序8-9比较)。

在进一步讨论类的设计时，我们还会讨论将职责推到服务器类。在本节中，我们将讨论允许类的设计人员对类的数据成员和成员函数的访问进行控制的技术。

图9-2描述了类Cylinder以及它与main()函数之间的关系。类有三个组成部分：数据、函数和区分类内外的界限。它显示了数据是在类内的，而函数一部分在类内(函数的实现)，一部分在类外(函数的界面对客户是可见的)。

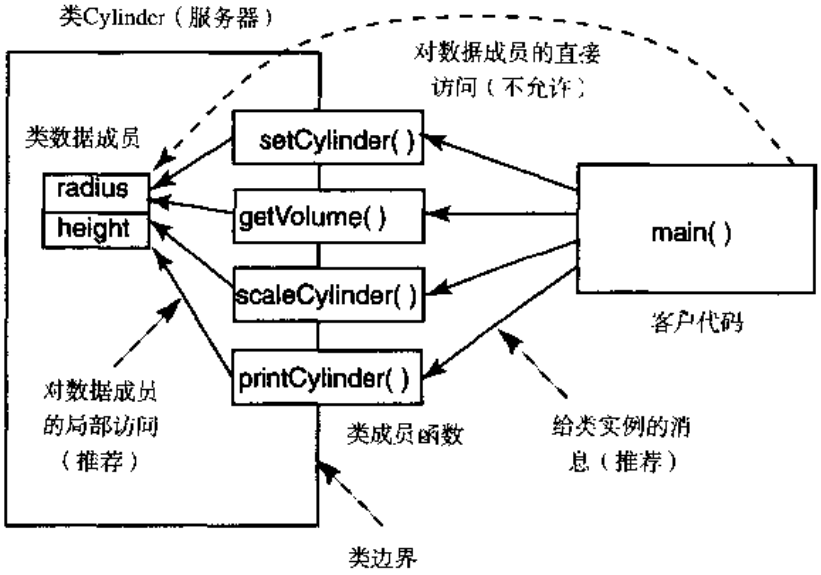


图9-2 类Cylinder与它的客户main()之间的关系

该图还显示了当客户代码需要Cylinder域的值时(例如为了计算圆柱体的体积、放大、打印或者设置域的值)，客户代码需要使用成员函数getVolume()、scaleCylinder()等，而不是直接访问域radius和height的值。这正是图9-2中虚线的意义，它表示对数据的直接访问是不允许的。

防止直接访问数据成员有两个原因。其一是限制类数据设计的修改对程序的影响。如果成员函数的界面保持相同(在类内的数据的设计改动时保持函数接口不变通常并不困难)，那么，我们只要改变成员函数的实现代码，而不需要修改客户代码。这一点对于维护来说是很

重要的。需要修改的函数集合定义良好，它们都被列举在类的定义中，而不需要察看程序的其他部分寻找其他可能的含义。

原因之二是将客户代码表示成一系列的成员函数调用，要比表示成对域值的具体计算更加容易理解。其前提是将职责推到成员函数，由成员函数负责为客户完成工作，而不仅仅是提供对域值的获取和设置，例如getHeight()、setHeight()函数。

为了获得这些益处，类内的所有数据应该是该类私有的，不能从外界直接访问，而只留下类外界可以访问的函数接口为公共的。这样可以防止客户代码建立和服务器类数据的依赖性。要记住依赖性一词在程序设计中是令人深恶痛绝的。程序代码不同部分之间的依赖性意味着：

- 程序开发过程中，程序员间需要更多的协作。
- 程序维护过程中，需要研究或者修改更多的代码。
- 在相同或相似的项目中难以重用代码。

同时，程序9-2的类的设计并没有加强对数据访问的任何保护。客户代码可以访问Cylinder对象实例的域，可以建立对Cylinder数据设计的依赖性，从而丧失了使用类的主要优势。

```
Cylinder c1, c2;                                    // define program data
c1.setCylinder(10,30); c2.setCylinder(20,30);       // use access function
c1.radius = 10; c1.height = 20;  . . .              // this is still ok!
```

C++允许类的设计者对类成员的访问权限进行很好的控制。我们可以使用关键字public、private和protected对类的每一个成员（数据或者函数）指定访问权限。下面是另一个版本的Cylinder。

```
struct Cylinder {                                   // start of class scope
  private:
    double radius, height;                          // data is private
  public:                                           // operations are public
    void setCylinder(double r, double h);
    double getVolume();                             // compute volume
    void scaleCylinder(double factor);
    void printCylinder();                           // print object state
} ;                                                 // end of class scope
```

这些关键字把类的作用域分段描述。例如，关键字private后面的所有数据成员或成员函数都拥有同样的私有的访问方式。在本例中，数据成员radius和height都是private访问控制方式，而所有的成员函数都是public方式。

一个类中可以按任何顺序放置任意数目的private、protected和public区段。在接下来的例子中，我们定义数据成员radius为private，两个成员函数setCylinder()和getVolume()为public，再定义数据成员height为private，两个成员函数scaleCylinder和printCylinder为public。

```
struct Cylinder {                                   // start of class scope
  private:
    double radius;                                  // data is private
  public:                                           // operations are public
    void setCylinder(double r, double h);
    double getVolume();
```

```
  private:
    double height;                          // data is private
  public:                                   // operations are public
    void scaleCylinder(double factor);
    void printCylinder();                   // print object state
} ;                                         // end of class scope
```

这样做的灵活性很大，但是程序员通常将使用相同的访问方式的所有类成员放在同一个区段内。

一般说来，在public区段内定义的类成员（包括数据成员和成员函数）都如前面的例子所示，可以被程序的其他部分访问。

而在private区段内定义的类成员（同样包括数据成员和成员函数）只能被该类的成员函数访问。（也可以被有friend访问权限的函数访问。我们将在第10章讨论friend。）在该类（或者友元）的作用域外使用私有的类成员的名字会导致语法错误。

在C++传统的类设计中，数据成员被定义为私有的，而成员函数被定义为公共的。

在protected区段内的类成员可以被该类的成员函数、以及从这个类（直接或间接）派生的类的成员函数访问。现在讨论继承的问题将偏离类的语法这一主题，我们将会在后面讨论继承。

客户函数（全局函数或者其他类的成员函数）只能通过公共的成员函数（如果有的话）访问私有的类成员。

```
Cylinder c1, c2;                                    // define program data
  c1.setCylinder(10,30); c2.setCylinder(20,30);     // use access function
//  c1.radius = 10; c1.height = 20;                 // this is now a syntax error
  if (c1.getVolume() < c2.getVolume())              // another access function
    c1.scaleCylinder(1.2);                          // scale it up
```

应该为类的客户提供必要的数据访问，并避免多余的访问，这些都是类的设计人员的责任。如果客户代码使用了不必要使用的类特性，就会建立额外的依赖性。如果这些特性被修改，客户代码也会受到影响。而且，类公共的特性越多，客户端代码程序员和维护人员为了有效地使用该类而需要知道的信息就越多。

将类的数据成员的访问控制方式设置为私有后，类Cylinder的实现细节就隐藏起来。如果Cylinder域的名字或者类型发生变化，只要Cylinder的类界面保持相同，客户代码就不会受到影响。这样可以防止客户代码建立对类Cylinder的依赖性。客户端代码程序员（和维护人员）就不必去学习类Cylinder的数据设计。

通常来说，容易改进的是数据部分。因此，一个典型的类通常把数据成员定义为private方式，把成员函数定义为public。这增加了程序的可修改性和类设计的重用性。注意，类的成员函数（无论是public或者private）都可以访问同一个类中的数据成员，无论它是public还是private。

正因为如此，访问同一数据集合的函数组应该被绑定在一起作为类的成员函数，而在客户代码中对这些函数的调用应该作为对类实例发送的消息。这样做增加了可重用性。

类是和程序的其他部分分隔开来的。它的私有部分不能被其他代码访问（就像一个函数或语句块中的局部变量一样）。

这个特性减少了设计小组成员之间的协作工作量，减少了小组成员之间交流可能造成的误解，进而提高了程序的质量。

在前面的所有例子中，都使用struct来定义C++的类。C++还允许使用关键字class来定义类。下面就是一个使用class而不是struct来定义类Cylinder的例子。

```
class Cylinder {                        // change from 'struct' to 'class' keyword
  private:
    double radius, height;              // data is still private
  public:                               // operations are public
    void setCylinder(double r, double h);
    double getVolume();
    void scaleCylinder(double factor);
    void printCylinder();
} ;                                     // end of class scope
```

这个类定义和前一个类定义有什么不同呢？没有，它们定义的类是完全相同的。这些类的对象也是一样的，没有丝毫的不同。在C++中使用关键字struct和class只有两点不同一个不同是关键字struct在C++中只有一种含义，只出于一个目的被使用（即如前一例子所示在程序中引入程序员定义类型）。另外一点不同是struct和class有不同的缺省访问控制方式，struct中（或者union）的成员缺省访问方式是public，而class中的缺省访问方式是private。除此之外，两者再没有其他的不同。

使用缺省的访问权限允许程序员按不同的排列顺序组织数据域和成员函数。一些程序员批评说先描述数据而不是函数的类定义是伪善的（正如前面的例子所示），在下面的例子中，我们会驳斥这种说法。类构造的目的就是要向客户代码隐藏数据的设计，在类规格说明的开头就描述所谓的“被隐藏”的数据似乎不太好。客户代码使用的是公共的成员函数，因此，在类的规格说明中首先列举这些函数较为合适。

```
struct Cylinder {          // some prefer to list public members first
    void setCylinder(double r, double h);  // operations are public
    double getVolume();
    void scaleCylinder(double factor);
    void printCylinder();
  private:
    double radius, height;                 // data is private
  } ;                                      // end of class scope
```

其他程序员认为了解所操作的数据对于理解成员函数的功能很重要。因此先描述数据并没有什么问题，毕竟数据隐藏并不像军事类型的加密信息，也不像不能为人所知的国家秘密。在程序设计中，信息隐藏和封装只不过是防止客户代码使用设计客户时的信息，而不是防止客户知道这些信息。因此，如果希望使用缺省的访问控制方式，使用关键字class比使用struct合适。

```
class Cylinder {                    // some prefer to list data first
   double radius, height;           // data is still private
  public:                           // operations are public
   void setCylinder(double r, double h);
   double getVolume();
   void scaleCylinder(double factor);
   void printCylinder();
} ;                                 // end of class scope
```

有些程序员认为使用关键字struct的档次比class的低，因为如果我们使用缺省的访问控制方式定义类，那么客户代码使用数据时没有任何的保护，这样就有可能破坏数据的封装性。

```
struct Cylinder {                   // default access rights are used
```

```
    double radius, height;              // data is not protected from client access
  void setCylinder(double r, double h); // methods are public
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                     // end of class scope
```

这的确破坏了数据封装性。但这并不能说明使用关键字struct比class低级。如果在这个设计中使用class代替struct，结果会更糟糕。明白是为什么吗?

```
class Cylinder {                        // default access rights are used
    double radius, height;              // data is protected from client access
  void setCylinder(double r, double h); // methods are not accessible
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                     // end of class scope
```

这样的类根本不能使用。是的，数据域现在是私有的（这很好），但是成员函数也是私有的，客户代码不能访问它们。这不是一个好的设计。

不依靠缺省的访问控制方式而是显式地进行指定可能会更好一些。

9.3 对象实例的初始化

当编译程序处理一个变量的定义时，它将使用其类型定义分配所需的内存空间。如果是static变量或者extern变量或者是动态变量，将会在内存的堆中分配空间；如果是局部自动类型则会使用栈stack。

对于简单变量、数组、结构或者有成员函数的对象来说，内存分配都是如此。如果后面的代码给变量赋值，变量就不需要在定义时被初始化。如果算法将变量作为右操作数使用，就需要初始化它的数据成员。

```
Cylinder c1;                    // data members are not initialized
double vol = c1.getVolume();    // no, this is no good
```

然而，如果在计算时可以使用一些缺省值，这种编码模式可能是合适的。但是，C++只初始化静态变量或者全局变量（缺省值是合适类型的0）。动态变量和自动变量没有初始值。

有时候我们希望指定缺省的初始值。如果可以像一般的变量那样在定义时初始化数据成员就好了，但在C++中数据成员的定义不能包含初始化操作。

```
class Cylinder {
  double radius = 100, height = 0; ...   // no, this is illegal in C++
```

类可以提供一个成员函数让客户代码调用它来指定对象的初始状态。

```
class Cylinder {
  double radius, height;
public:
  void setCylinder(double r, double h);  ... } ;
```

有了这个函数，客户代码可以将消息setCylinder()发送给Cylinder对象。

```
Cylinder c1;
c1.SetCylinder(100.0,0.0);      // set radius to 100, height to zero
```

这样当然就可以纠正了。这段代码允许我们指定任何初始值而不是指定缺省值。这种情况下使用类的构造函数就有用多了。

9.3.1 作为成员函数的构造函数

可以使用构造函数隐式地初始化类对象。构造函数也是类的成员函数，但语法上要求比其他成员函数更严格。不能为构造函数任意命名，我们应该为构造函数指定与类名相同的函数名。构造函数的界面不能有返回值，就是void类型也不行。即使构造函数包含return语句也不能返回任何值。

```
class Cylinder {
  double radius, height;
public:
  Cylinder ()                    // same name as class, no return type
    { radius=1.0;  height=0.0; }  // no return statement
    ... };
```

当客户代码创建一个对象时，将会调用缺省构造函数（default constructor）。

```
Cylinder c1;                     // default constructor: no parameters
```

之所以称为缺省构造函数，是因为它没有参数。尽管听起来有点奇怪，但事实上是这样。构造函数并不能像其他成员函数一样被显式地随便调用。

```
c1.Cylinder();      // syntax error: no explicit calls to constructors
```

构造函数只是在对象被创建的时候调用，以后都不会被调用。编译程序在对象实例被创建后立刻产生隐式地调用构造函数的代码。因此，构造函数通常放在类规格说明中的公共区段部分。否则，试图创建类实例将会产生与访问私有的类成员相同的错误。

一般说来，对象的实例可以如下方式创建：

- 在程序开始时（extern或者static对象）。
- 进入包含对象定义的作用域入口处（自动类型对象）。
- 当一个对象作为参数按值传递给函数（或者从函数返回）时。
- 当使用new运算符（而不是malloc）动态创建变量时。

现在我们知道为什么构造函数不能有返回值了，因为它是由编译程序产生的代码隐式地调用的，没有什么地方需要使用这个返回值。

像成员函数一样，构造函数可以有参数；因此，构造函数可以被重载。如果必要的话，构造函数可以有缺省值。当一个类有多个构造函数时，在对象被创建时可以调用其中任何一个。但究竟调用哪个构造函数，依赖于上下文即客户代码在创建对象时提供的参数集合（参数个数和类型）。

在类中提供构造函数意味着为类的客户提供服务，因为客户端代码程序员不再需要显式地调用初始化函数。但我们要注意为构造函数提供参数。下面是有两个参数的构造函数例子。

```
class Cylinder {
  double radius, height;                  // initialized in constructors
public:
  Cylinder(double r, double h);           // member function prototype
  void setCylinder(double r, double h);
  . . . . . } ;
```

```
Cylinder::Cylinder(double r, double h)   // scope operator
   { radius = r;  height = h; }
```

注意在类边界外实现的构造函数名。第一个Cylinder表明该成员函数所属的类，第二个Cylinder表明该成员函数名（与类名相同）。少于两个参数的构造函数有特别的函数名（很快就会看到）。有两个或者两个以上参数的构造函数没有特别的名字，它们只是一般的构造函数而已。

有两个参数的这个构造函数和setCylinder()所做的工作是一样的，都是将客户所提供的参数值赋给数据成员。不同的地方在于setCylinder()可以在客户代码中为相同的对象实例多次调用；而构造函数只能在创建对象时被调用一次。

下面是在客户代码中激活构造函数的一些例子，它们使用不同的语法形式调用有两个参数的构造函数。注意第二个语句使用的赋值运算符并不表示执行了赋值操作。虽然外表看起来是赋值，但事实并非如此，而只是构造函数调用的一个不同语法形式。

```
Cylinder c1(3.0,5.0);       // a constructor call for a named object
Cylinder c2 = Cylinder(3,5);       // it is still a constructor call
Cylinder *r = new Cylinder(3.0,5.0);       // unnamed object
```

注意有参数变量的语法。这是一个新语法。C++语言设计有一个大胆的目标是，将基本类型和程序员定义类型变量统一对待。对于基本类型，我们使用赋值运算符进行初始化。有了程序员定义类型，也可以像类对象那样使用带参数的语法初始化基本类型的变量。

```
int x1(20);                  // same as int x1=20
```

当通过malloc()为对象分配空间时，不会调用构造函数。因此，客户代码必须显式地初始化类对象。

```
Cylinder *p = (Cylinder*)malloc(sizeof(Cylinder));       // no constructor call
p->setCylinder(3,5);                          // object fields are assigned values
```

调用malloc()是C++中创建对象而不调用构造函数的惟一方法。创建所有其他命名的和动态的对象都会调用构造函数。从现在开始，不可能出现只是创建一个对象实例并分配空间的情况，因为创建任何对象都会调用构造函数。这再次需要我们在思想上有所改变。每次看到对象实例被创建时，都应该提醒自己：这意味着调用了一个构造函数，调用的是哪一个构造函数呢？

9.3.2 缺省构造函数

许多类无需构造函数，因为类对象并不需要缺省初始化。当类的设计人员没有为类编写构造函数时，系统将会为该类提供一个（什么也不做的）缺省构造函数。

```
class Cylinder {                  // OK if no constructors/destructors
     double radius, height;  // data is protected from client access
public:
  void setCylinder(double r, double h);    // methods are accessible
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                          // end of class scope
```

在前面部分我们讨论的所有版本的类Cylinder都是使用系统提供的缺省构造函数。当

时没有提及这个问题是为了避免不必要的复杂性。客户代码创建Cylinder对象时，将会调用缺省构造函数。

```
Cylinder c1;     // default constructor is called, no initialization
```

既然缺省构造函数不干任何的事情，我们为什么还要去了解它呢？但是我们需要知道的是，如果类定义了一个非缺省的构造函数（即有参数的构造函数），系统将不会再提供缺省的构造函数。

为什么我们要知道这一点呢？因为如果客户设计人员定义了需要缺省构造函数的类变量和数组，就会出现语法错误。

上一个版本的Cylinder类没有程序员定义的构造函数。所以系统提供了一个不做任何事情的缺省构造函数。当变量c1被创建时，构造函数将会被调用。我们从何得知的呢？因为必须调用某个构造函数。（不会有不调用构造函数就创建对象的事情。）调用哪一个构造函数呢？这依赖于所提供的参数个数。变量c1没有提供任何参数，这就证明无参数的构造函数会被调用。无参数的构造函数就是缺省构造函数。那么类定义提供了缺省构造函数吗？没有。类定义提供了任何构造函数吗？也没有。因此系统将会为之提供一个缺省构造函数，它将不做任何事情。

下面让我们来看另一个版本的类Cylinder，它提供了一个程序员定义的一般构造函数。这意味着系统不再提供缺省构造函数。

```
class Cylinder {
     double radius, height;
  public:
     Cylinder(double r, double h)          // this is not enough
     { radius = r;  height = h; }
     . . .  } ;
```

当客户代码试图创建Cylinder对象时，就会出现如下问题：

```
Cylinder c1(3.0,5.0);                 // this is OK
Cylinder c2, c[1000];                 // 1001 syntax errors
Cylinder *p = new Cylinder;           // one syntax error
```

这里，我们要创建1001个Cylinder对象实例，但没有提供任何参数。还记得不可能创建一个对象而不调用构造函数吗？因此，编译程序尝试为1001个缺省构造函数的调用产生代码。调用哪一个构造函数呢？因为并没有指定任何参数，编译程序试图调用没有参数的构造函数即缺省构造函数Cylinder::Cylinder()。但是这个版本的Cylinder类没有定义缺省构造函数。又因为它定义了一般构造函数，所以系统没有提供缺省构造函数。客户代码调用成员函数1001次以初始化1001个Cylinder对象时会有什么后果呢？既然在Cylinder的类规格说明中找不到这个函数，编译程序会产生语法错误。对此，我们一定要弄清楚其中的逻辑。

如果我们在类定义中加上一个缺省构造函数，这个问题将得到解决。这个缺省构造函数可以像系统提供的缺省构造函数一样什么都不干，也可以将对象的数据成员赋以合理的值。

```
class Cylinder {
     double radius, height;
  public:
     Cylinder ()           // programmer-supplied default constructor
     { radius = 100.0; height = 0.0; }     // reasonable values
     Cylinder(double r, double h)          // general constructor
```

```
{ radius = r;  height = h; }
. . .  } ;
```

客户代码：

```
Cylinder c1(3.0,5.0);                    // this is OK
Cylinder c2, c[1000];                    // now this is OK, too
Cylinder *p = new Cylinder;              // no syntax error
```

注意每个对象被创建的时候，至少有一个函数被调用。在上面的代码中，构造函数是内联函数，构造函数也可能有性能问题。在C++中不存在创建对象而不调用函数的情况。

注意 在C++中，创建对象后总是会紧跟着一个函数调用。如果类没有定义构造函数，对象创建后会调用系统提供的缺省的构造函数。如果类中定义了任何构造函数，系统将不再提供缺省的构造函数。在这种情况下，我们必须提供参数来创建对象数组或对象，因为系统撤销了它所提供的构造函数。

9.3.3 拷贝构造函数

我们要着重指出，C++关于对象哲学的一个重要思想是：类是类型。为程序定义类扩展了系统的基本C++类型。C++也希望将程序员定义类型视为基本类型对待。

例如，我们可以声明基本类型的变量，而不指定它们的初始值。因此，我们也可以这样处理对象变量。

```
int x; Cylinder c1;                      // noninitialized variables
```

它们的语法是一样的，但意义不同。基本类型变量的定义只是为其分配了内存空间；而程序员定义类的变量的定义除了为变量分配空间外，还调用了缺省构造函数。如果该类没有定义构造函数，这个缺省的构造函数将由系统提供，并且不做任何操作。如果类定义了构造函数，如上所示那样定义类变量将会导致语法错误，除非类也定义了缺省的构造函数。自定义的缺省构造函数可以什么事情都不做，也可以将对象的域初始化为缺省值。

类似地，我们有时候可能会用同类型的另一个变量来初始化一个基本类型的非类变量。C++提供相似的语法允许客户代码用同一个类的另一个对象初始化该类的一个对象。

```
int x(20); Cylinder c1(50,70);   // objects are created, initialized
int y=x; Cylinder c2=c1;       // initialization from existing objects
```

不要被第二行中的赋值运算符所误导。这些语句中并没有赋值运算。赋值运算符在这里重用来标识初始化。要记住，当类型名出现在变量名的左边时，我们处理的是初始化工作；当没有类型名而变量名单独出现时，我们处理的才是赋值运算。为什么我们要分清楚这些细小的差别呢？接下来我们就会知道在这两种不同的情形下所调用的函数是不同的。

在本例中调用的是什么函数呢？答案是简单的。因为创建和初始化了对象，所以这里调用的是构造函数。哪一个构造函数呢？正如我们在前面提到的，这依赖于上下文，即对象创建时所提供的实际参数个数和类型。

在上面的例子中，我们用对象c1作为一个参数初始化对象c2，对象c1的类型是Cylinder。因此，被调用的是只有一个参数且其类型为Cylinder的构造函数。这种推导过程清楚吗？我们应该在每次分析对象创建语句时都做这样的推导。

只有一个与该类同类型的参数的构造函数称为拷贝构造函数（copy constructor），它是一

种特殊的构造函数。之所以这样命名，是因为它把已存在的源对象中的成员值复制到刚刚创建的目标对象的域中。在前面的例子我们看到，类Cylinder并没有参数类型为Cylinder的构造函数，而只是有一个具有两个double参数的一般构造函数和一个没参数的缺省构造函数。这是否意味上面的语句是错误的呢？不会，这再次证明了学习C++的丰富多彩。

如果一个类没有定义构造函数，C++会为之提供自己的拷贝构造函数。这个构造函数将源对象中的数据成员值一位位地复制到目标对象中。与系统提供的缺省构造函数不同的是，即使类定义了其他构造函数，系统提供的拷贝构造函数也不会被撤销。因此，我们可以认为它一直存在。

像Cylinder这样的类，定义程序员定义的拷贝构造函数没有什么意义。因为我们在这个拷贝构造函数中能做的只不过是复制参数的radius和height的值，而这正是系统提供的拷贝构造函数完成的功能。使用程序员定义的拷贝构造函数的惟一原因是为了调试程序。

```
class Cylinder {
  double radius, height;
public:
     Cylinder (const Cylinder &c)
     { radius = c.radius;  height = c.height;
       cout << "Copy constructor: " << radius << ", "
. . . . } ; << height << endl; }
```

注意，参数必须是给定类型的变量的引用而不是给定类型的变量。如果拷贝构造函数的参数是按值传递会出现什么后果呢？

```
Cylinder (Cylinder c)              // incorrect constructor interface
  { radius = c.radius;  height = c.height;
    cout <<"Copy constructor: "<< radius <<", " <<height <<endl; }
```

当调用这个构造函数时，我们将为实际参数创建一个副本——即给一个Cylinder变量分配空间，并用实际参数域的值进行初始化。但仔细想想怎么样呢？在C++中对象创建时一定会调用构造函数！“给一个Cylinder变量分配空间，并用实际参数域的值初始化”意味着要为拷贝构造函数的实际参数调用拷贝构造函数。然后这第二个拷贝构造函数被调用时，又会创建它的实际参数的副本，再次调用构造函数……。这种递归调用过程将一直持续到用户失去了等待的耐心或机器的栈空间溢出为止。

如果因为对递归调用没有足够的经验而对上面的描述是一知半解，可以试验一下按值调用一个拷贝构造函数的参数，看看其结果，可以肯定大家不会再这样做了。我们在此还是强调一下这个问题。

警告　拷贝构造函数只有一个参数，其类型就是该构造函数所属的类类型。一定要按const引用传递这个参数而不能按值传递。按值传递拷贝构造函数的参数会导致无穷无尽的拷贝构造函数递归调用。

对拷贝构造函数还有一点需要说明。因为它是一个函数调用，我们可以使用和调用一般构造函数一样的标准语法。

```
int x = 20; Cylinder c1(50,70); // objects are created, initialized
int y=x; Cylinder c2(c1);       // call to Cylinder copy constructor
```

但是，C++想要以相同的方式对待对象和基本类型的变量。这意味着构造函数调用的初始

化语法也向后扩展到基本类型变量上，即使这些基本类型变量并没有可以调用的构造函数。这种语法只能用于C++而不能用于C。

```
int x(20);                    // object is created and initialized
int y(x);                     // variable y is created and initialized
```

还有一个关于构造函数调用的通用说明。除了缺省构造函数外，普通函数调用（及其括号）的语法都可以运用在所有的构造函数上。下面是对命名的变量和对动态变量使用一般构造函数和拷贝构造函数的例子。

```
Cylinder c1(50,70);                 // general constructor is called
Cylinder c2=c1;                     // copy constructor is called
Cylinder *p = new Cylinder(50,70); // general constructor is called
Cylinder *q = new Cylinder(*p);     // copy constructor is called
```

这样的语法不能应用于缺省构造函数。如果客户代码调用缺省构造函数时使用了括号，将会导致语法错误。

```
Cylinder c1();                      // syntax error
Cylinder c2;                        // default constructor is called
Cylinder *p = new Cylinder();       // syntax error: parentheses
Cylinder *q = new Cylinder;         // default constructor is called
```

为什么会出现这种不一致呢？这是为了方便编写编译程序的代码。我们可以看上面一段代码中的第一行，我们怎么知道它是一个构造函数的调用，还是一个名为c1()的返回类型为Cylinder的函数原型呢？单从字面上来看，无论是编译程序的编写者还是C++程序员都无法区分。避免这种二义性的一个方法是，规定函数原型必须在源代码文件的开始处而不能在其他地方使用。这个建议听起来不错，因为我们通常都是把函数原型放在文件开头的。然而，C语言允许我们在文件中的任何位置使用函数原型，而C++的设计十分看重向后与C语言兼容，所以不可能让文件其他位置出现的函数原型成为语法错误。Java语言并没有考虑向后与C语言兼容，所以，使用Java在客户代码中调用缺省构造函数的语法和调用所有其他构造函数的语法是一样的。

9.3.4 转换构造函数

如果一个类的构造函数只有一个参数，而且这个参数不是该类的相同类型而是其他类型，那么这个构造函数被称为转换构造函数。如果客户代码想在创建每个对象时只指定某一个数据成员的值，而让其他域的值使用相同的缺省值，转换构造函数就很有用。

例如，在一个建模程序中，我们可能使用不同的半径值创建多个Cylinder对象。最开始时，所有的对象都应该设其高度值为0，然后随着建模过程（例如生长和动脉测量、与电子元件连接、通过管道壁进行热交换等）的进行而慢慢增高。

```
Cylinder c1(50.0);          // conversion constructor is called
Cylinder c2 = 30.0;         // conversion constructor is called
```

这些语句虽然语法形式不同，但意义相同，都是调用转换构造函数。

与缺省构造函数和拷贝构造函数不同，系统没有提供转换构造函数。除非在类中定义了有一个双精度类型参数的转换构造函数，否则上面的语句是错误的。转换构造函数指定如何处理所提供的惟一值，以及为对象的其他域使用什么缺省值。在接下来的例子中，类Cylinder定义了4个构造函数：缺省构造函数、拷贝构造函数、转换构造函数和有两个参数

的一般构造函数。

```
class Cylinder {
    double radius, height;
  public:
  Cylinder ()              // programmer-supplied default constructor
    { radius = 1.0; height = 0.0; }
  Cylinder (const Cylinder &c)        // copy constructor
    { radius = c.radius;  height = c.height; }
  Cylinder(double r, double h)
    { radius = r;  height = h; }      // general constructor
  Cylinder (double r)
    { radius = r;  height = 0.0; }  // conversion constructor
  . . . } ;
```

转换构造函数与C++的强类型系统相抵触。正如前面提及的，所有的现代高级语言都支持强类型。如果在某特定的上下文中需要一个类型的值，提供另一个类型的值将产生语法错误。例如下面这段代码：

```
Cylinder c2 = 30.0;          // conversion constructor is called
```

如果Cylinder是一个简单的C语言结构类型，这将会导致语法错误；如果Cylinder是一个没有转换构造函数的类，同样会产生语法错误；在这两种情况下我们都不必运行程序和分析程序输出就知道是什么出错。如果Cylinder是一个有转换构造函数的类，就没有语法错误。如果我们是特定这样做的，那很好，如果我们是误操作，友好的编译程序不会防止我们犯这样的错误。这种情况下，强类型系统被削弱了。

还有一个例子，考虑本章的CopyData()函数（再次假设数据成员radius和height是公共的）。

```
void CopyData(Cylinder *to, const Cylinder &from)          // copy Cylinder data
  { to->radius=from.radius;  to->height=from.height; }     // arrow notation
```

如果Cylinder是一个简单的C结构类型或者是一个没有转换构造函数的C++类，在客户代码中调用这个函数会导致语法错误：

```
CopyData(&c2,70.0);          // the FROM Cylinder is missing here
```

但如果Cylinder中有转换构造函数，编译程序将会产生代码创建一个临时的未命名的Cylinder对象，并为这个临时对象调用转换构造函数（使用70.0作实际参数），再把这个未命名的临时对象作为第二个参数传递给CopyData()。

如果客户代码使用的不是double类型的数值类型值也没有问题。编译程序会产生代码将这个数值转换为double，然后再将这个转换后的值作为实际参数调用转换构造函数。

```
Cylinder c2 = 30;            // 30 is converted to double
CopyData(&c2,70);            // 70 is converted to double
```

当然，如果这段客户代码确实是我们所需要的，那么C++提供的这么灵活的能够实现我们意图的功能真是太好了。但是，如果我们误写了这段代码，可惜的是编译程序不会告诉我们有错误，因此我们不能在程序运行前更正它。

9.3.5　析构函数

一个C++的对象或者在程序执行结束时撤销（对extern和static对象而言），或者从某

作用域的闭花括号退出时撤销（自动类型对象），或者在执行delete运算符时撤销（由new创建的动态对象），又或者在调用库函数free()时撤销（用malloc()分配内存的对象）。

无论对象何时被撤销（除使用free()函数外），在撤销前都会立刻调用类的析构函数。如果类没有定义析构函数，将调用系统提供的缺省析构函数（和缺省构造函数一样，这个析构函数也什么都不干）。

程序员提供的析构函数和构造函数相似，也是一个类的成员函数。析构函数的语法比构造函数的语法还要严格，在函数接口不允许有返回值，在函数体内也不能使用return语句。析构函数的函数名和类名一样，只不过在类名前加上“~”，例如~Cylinder()。但析构函数和构造函数不同的是它不允许有参数。

类构造函数和析构函数都是放置调试所用的打印语句的好地方。

```
class Cylinder {
  double radius, height;
public:
  ~Cylinder ( )          // programmer-defined destructor: no return type
  { cout << "Cylinder (" << radius << ", " << height
         << ") is destroyed" << endl; }     // no return value
  . . . . } ;
```

当析构函数在类的作用域以外实现时，要使用作用域运算符。注意“~”是函数名的一部分，而不是作用域运算符的一部分。

```
Cylinder::~Cylinder ( )          // class destructor: no return type
  { cout << "Cylinder (" << radius << ", " << height
         << ") is destroyed" << endl; }     // no return value
```

析构函数不能有参数，所以析构函数不能被重载，因为重载函数必须有不同的参数列表。所以一个类最多只能有一个析构函数。

如果对象使用了动态内存或者其他资源（如文件、数据库等），就必须提供程序员定义的析构函数。析构函数应该将这些资源返回给系统以避免资源泄漏。析构函数所做的通常和构造函数所做的功能相对应，如内存空间的分配和撤销、文件的打开与关闭等。

我们来看一个析构函数很有用的例子。类Name为包含人名的字符串分配了空间。构造函数初始化一个字符串。（这是个转换构造函数，因为有一个不同于类Name类型的参数。）为简单起见，所有的数据都是公共的，并只提供一个方法show_name()，它在屏幕上显示对象的内容。

```
struct Name {
     char contents[30];        // fixed size object, public data
  Name (char* name);           // or Name(char name []);
  void show_name();
  } ;                          // destructor is not needed yet

Name::Name(char* name)         // conversion constructor
{ strcpy(contents, name); }    // standard action: copy argument data

void Name::show_name()
{ cout << contents << "\n"; }
```

客户代码可以定义这个类型的变量，并把对象的内容显示在屏幕上。

```
Name n1("Jones");                  // conversion constructor is called
```

```
Name *p = new Name("Smith");      // conversion constructor is called
n1.show_name();  p->show_name();
delete p;                         // unnamed object is deleted
```

无论名字的内容有多大，这种类的设计都分配了相同的内存空间。如果名字太短，会造成空间的浪费；但如果名字太长又会耗用内存。

动态内存管理是解决这个问题的常用方法。类不再拥有一个固定长度的字符数组作为数据成员，而是定义一个字符指针。堆内存的大小将依赖于客户代码提供的名字的长度。在构造函数中，调用strlen()来计算所需的堆内存大小（额外的字符是结束符0），然后分配内存，并调用strcpy()初始化这个堆内存。

```
struct Name {
     char *contents;                    // pointer to dynamic memory: still public
   Name (char* name);                   // or Name(char name []);
   void show_name();
     } ;  // destructor is needed now

Name::Name(char* name)                  // conversion constructor
{ int len = strlen(name);               // number of characters in argument
   contents = new char[len+1];          // allocate heap memory for argument data
   if (contents == NULL)                // 'new' was not successful
     { cout << "Out of memory\n";  exit(1); }      // then give up
   strcpy(contents, name); }            // success: copy argument data

void Name::show_name()
{ cout <<contents << "\n"; }
```

我们把客户代码放在全局函数Client()中，以讨论当使用新版本的Name类时会发生什么情况。

```
void Client()
{ Name n1("Jones");                  // conversion constructor is called
   Name *p = new Name("Smith");      // conversion constructor is called
   n1.show_name();  p->show_name();
   delete p;                         // destructor for object pointer by p is called
   }                           // p is deleted, destructor for object n1 is called
```

当执行到Client()函数中的delete p;语句时，释放指针p所指向的内存。这段内存只包含指针contents而已，而指针contents所指向的内存并没有被释放，也没有变成不可访问。这就是内存泄漏。注意语句delete p;并不是删除指针p，而是删除p所指向的内存。按照作用域规则，指针p本身是在它定义时所在的作用域终止时删除的。即在函数Client()执行到结束的花括号处被删除。

同样，Client()终止时，局部对象n1被撤销，其数据成员contents指针被还回到栈中。而指针contents所指向的内存并没有还回给系统，因此也会产生内存泄漏。

正是对于这种动态管理其资源的类类型，析构函数才显得尤为重要。需要析构函数来维护C++程序的完整性。每当对象按作用域规则撤销或者被运算符delete撤销时（但并不是被调用函数free()撤销），析构函数都会被调用。因此，析构函数是一个释放该对象在其生存期内所用的内存空间（和其他资源）的好地方。（虽然大多数情况下是构造函数分配动态内存，但是其他成员函数也可以这么做。）

类Name的析构函数很简单。程序9-3的类Name有一个还回堆内存的析构函数。图9-3显示

了程序的运行结果，有构造函数和析构函数的调试信息。

程序9-3 使用析构函数还回为有名或无名对象分配的堆内存

```
#include <iostream>
using namespace std;

struct Name {
     char *contents;                          // public pointer to dynamic memory
   Name (char* name);                         // or Name (char name []);
   void show_name();
   ~Name(); } ;                               // destructor eliminates memory leak

Name::Name(char* name)                        // conversion constructor
{ int len = strlen(name);                     // number of characters
   contents = new char[len+1];                // allocate dynamic memory
   if (contents == NULL)                      // 'new' was not successful
     { cout << "Out of memory\n";  exit(1); }     // give up
   strcpy(contents, name);                    // standard set of actions
   cout << "object created: " << contents << endl; }     // debugging
void Name::show_name()
{ cout <<contents << "\n"; }

Name::~Name()                          // destructor
{ cout << "object destroyed: " << contents << endl;     // debugging
   delete contents; }                 // delete heap memory, not pointer 'contents'

void Client()
{ Name n1("Jones");                    // conversion constructor is called
   Name *p = new Name("Smith");        // conversion constructor is called
   n1.show_name();  p->show_name();
   delete p;                    // destructor for object pointed to by p is called
   }                            // p is deleted, destructor for object n1 is called

int main()                             // pushing responsibility to server functions
{ Client();
   return 0;
   }
```

```
object created: Jones
object created: Smith
Jones
Smith
object destroyed: Smith
object destroyed: Jones
```

图9-3 程序9-3的输出结果

当函数Client()执行到delete p;语句时，类Name的析构函数被调用，并执行delete contents;语句。当Client()执行撤销对象n1时，析构函数被调用并执行delete contents;语句。这样就消除了内存泄漏问题。

图9-4显示了Client()函数的内存使用情况。图9-4a显示了有名对象n1和指针p所指向的无名对象创建后的内存状态，数字1到5说明了内存空间分配的先后顺序。首先为n1分配栈空间（根据作用域规则），接着分配"Jones"的堆空间（通过构造函数），然后分配指针p的栈空间（根据作用域规则），并分配（p所指向的）未命名对象的堆空间，最后分配"Smith"

的堆空间。

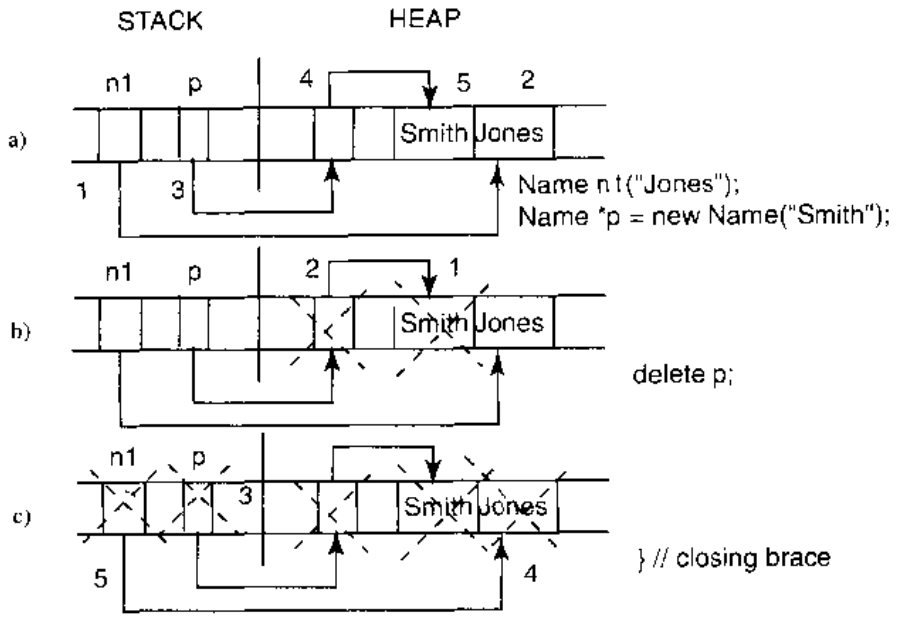


图9-4　程序9-3的Client()函数的内存管理图

图9-4b和图9-4c演示了对象的撤销过程。图9-4b显示了"Smith"所占用的堆空间最先被释放（通过析构函数），然后未命名的堆对象被删除（通过delete运算符）。因为delete运算符不删除指针p，而是删除指针p所指向的堆内存，所以此时指针p仍然存在。

图9-4c显示了作用域规则回收了指针p和命名对象n1的栈空间。撤销p本身没有导致任何事件发生。而对象n1的析构将会调用Name的构造函数，释放"Jones"所占用的堆空间（这段空间是构造函数分配的），然后释放对象n1本身所占用的栈空间。

一定要多花些时间认真琢磨图9-4，并自己编写代码仔细体会。一些程序员认为，如果将堆内存（本例中的"Smith"和"Jones"）看做Name对象实例的一部分，就比较容易分析。而我们认为将数据成员看做对象实例的一部分更加方便，可以认为堆内存是为每个对象实例分配的额外资源，最后需要还回给系统。从这个观点来看，为对象本身分配的空间是它的数据成员所定义的大小，而不是由它的构造函数的参数决定的。但这只不过是看待问题的角度问题。

注意，如果在函数Client()中没有delete p;语句，则指针p所指向的空间（包括它的指针contents和contents所指向的空间）就永远不会还回给系统。维护程序的完整性是客户端代码程序员的职责。对作用域规则控制的对象来说不需要使用delete运算符。例如，当执行到达函数Client()的闭花括号处时，对象n1会被自动删除。客户端代码程序员不需要对此进行操作。这种情况下进行内存管理，所需要的只是服务器程序员在类Name的设计中包含析构函数。

9.3.6　构造函数和析构函数的调用时间

术语“构造函数”暗示这个成员函数是用来构造一个对象的，而术语“析构函数”暗示这个成员函数是用来撤销一个对象的。事实上并非如此，术语并没有正确地描述构造函数和

析构函数执行的任务。

在前面的讨论中，我们很精确地指出：构造函数是在对象创建之后调用的，而析构函数则在对象撤销之前被调用。有关C++的书经常不会过多地注意这个问题，而只是说当对象被建立和撤销时会调用构造函数和析构函数。这种说法是令人遗憾的，因为给人的印象就是构造函数构造对象而析构函数撤销对象。

情况并非如此，事实上是作用域规则（对命名对象来说）和运算符new和delete（对未命名对象来说）建立与撤销了对象。构造函数只是在对象的域创建后才初始化这些域和分配额外的资源，例如堆内存。而析构函数只是还回对象在其生存期内所需的资源，例如在构造函数或其他函数中分配的堆资源。

总之，不是构造函数构造对象，也不是析构函数撤销对象。

9.3.7 类作用域和嵌套作用域中的名字覆盖

究竟在什么时候调用构造函数和析构函数，这依赖于作用域和对象实例的存储类别。

作用域描述了程序代码的不同部分对变量和对象的访问能力。存储类别描述了变量和对象从创建到撤销的生存期。这一节是对第6章有关作用域和存储类别讨论的扩展。如果觉得这一部分过于复杂，可以先跳过不看（但希望在以后回过头来看）。这一部分内容很重要，但是可以等大家积累了足够多的C++代码编写和阅读经验后再看。

因为全局变量可以在文件的任意位置定义，甚至可以放在函数定义之后，因此在文件中全局变量声明前定义的函数不能访问该全局变量。

```
Cylinder Cglobal;                    // available everywhere in the file

int main ()
{ Cylinder c;                        // scope is limited to main()
  . . . . }

int y;                               // not visible in main(), visible in foo()

void foo()                           // cannot be called from main() if no prototype
{ y = 0;                             // access to global variable
  Cylinder Clocal;
  Clocal.setCylinder(10,30);         // public members are visible
  Cglobal.setCylinder(5,20); }       // public members are visible
  . . . .                            // Cglobal, y, foo() are visible here
```

这是合法的C++代码，但不是好的编程风格。

因为局部变量可以在块（包括函数块和未命名块）的任何地方定义，因此它不能被该块中在变量定义前的代码访问。如果一个局部变量和全局变量同名，则在局部变量被定义的块中使用的是局部变量，在块外使用的是全局变量。

除了上面所说的两种作用域外（第6章有详细说明），C++还有一种作用域：类作用域。在类作用域内定义的所有名字（包括数据成员或成员函数，无论公共还是私有）在整个类作用域内是可见的。全局和局部作用域中应用的一遍编译（one-pass compilation）规则在此不适用。因此在所有的类Cylinder的例子中，无论成员的定义顺序如何，Cylinder的成员函数都可以访问Cylinder的数据成员。

如果类作用域中定义的名字和某个全局名相同，则在类中使用的这个命名都指的是类作

用域中定义的命名，在类外使用此命名则指的是全局命名。

如果类作用域中定义的某个命名和一个成员函数中定义的局部命名相同，则在这个成员函数里使用局部命名，而在其他成员函数中使用类作用域中定义的命名。

简单说来，局部命名会隐藏类作用域内的命名和全局命名，类作用域的命名会隐藏全局命名。而使用全局作用域运算符::（对全局命名而言）和类作用域运算符（对类作用域命名而言）可以忽视这些命名隐藏规则。

在下面的例子中，标识符radius分别用于全局变量、类Cylinder的一个数据成员和Cylinder成员函数setCylinder()的局部变量

```
double radius = 100;                    // global name

struct Cylinder {                       // start of the class scope
     double radius, height;             // member radius hides global radius

void setCylinder(double r, double h)
{ double radius;
  radius = r;  height = h;              // local radius hides data member radius
  Cylinder::radius = radius; }          // class scope operator overrides the rule

void scaleCylinder(double factor)
{ radius  = ::radius;                   // global scope operator overrides the rule
  height *= factor; }
  . . . } ;                             // end of class scope
```

setCylinder()成员函数的参数r的值被赋给radius时，实际上是函数内的局部变量被赋值，而不是Cylinder的数据成员被赋值。要想给数据成员radius赋值，应该使用类作用域运算符。在成员函数scaleCylinder()中，radius表示类数据成员，要想取得全局变量radius的值，应该使用全局作用域运算符。

有时候，某些程序员喜欢为方法的参数名和数据成员名设置同样的名字。例如下面的setCylinder()函数是不正确的

```
void Cylinder::setCylinder(double radius, double h)       // incorrect function
{ radius = radius;  height = h; }     // parameter is local, hides data member
```

这个函数可以通过编译，运行也没有任何问题。然而编译程序和类的设计人员对radius=radius;有不同的理解。对于设计人员来说，等号左边的radius表示数据成员radius，等号右边的radius则表示参数。对于编译程序来说，等号两边的radius都表示参数。虽然把参数值赋给它本身没有多大的意义，但编译程序并不会因而另外猜测程序员的意图。想要把参数赋值给它本身？没问题，在C++中是完全合法的。

存储类别指的是变量的生存期：变量何时被创建？何时被撤销？（automatic、external、static）。

局部自动变量在程序运行到其定义时在栈中分配内存（每次执行同一作用域，变量分配的内存位置会有不同）。如果不同作用域使用了相同名字的变量，这些变量指向不同的内存空间。如果没有初始化，该内存的内容是无定义的。对对象而言，空间一被分配就调用构造函数。

当执行到自动变量定义时所在的块结尾时，该变量被撤销。对对象而言，在空间被还回给系统之前立即调用析构函数。

对于外部变量和静态变量（不论是局部还是全局），程序开始执行前就为它们分配并初始化了固定的内存空间。如果没有显式地指定初始值，系统会用0进行初始化。对对象而言，构造函数的代码（以及构造函数可能调用的所有函数）在其空间被分配后main()开始前执行。不同对象的构造函数的调用顺序是未定义的。

当main()终止（即执行到其闭花括号处或以其他方式终止）时，外部和静态（不论是局部还是全局）变量被撤销。至于对象，在对象被撤销前调用析构函数。

这和我们在第6章中对有关存储类别的讨论有很大不同。如果程序没有使用任何程序员定义的类变量，代码的执行顺序很清晰：它从main()函数中的第一条语句开始到main()函数的最后一条语句。

显式地为动态变量分配空间和撤销空间，通常不在同一个函数（作用域）中调用new和delete运算符或者malloc()和free()函数。我们常常在一个函数中为动态变量分配内存，将内存依附在动态结构上（栈、队列、链表等），而在另一个函数中撤销该变量（这些函数应该可以属于同一个客户类。）

9.3.8 用运算符和函数调用的内存管理

在这一部分，我们将会把运算符new、delete和函数malloc()、free()进行比较。和前面那一节相类似，如果觉得这部分的内容太繁杂，可以跳过不看，但以后一定要回到这一部分学习。这有两方面的原因：首先是应该尽量使用new和delete，不要使用malloc()和free()；第二点是从本部分的例子中我们将看到什么是不符合面向对象编程思想的。

注意，构造函数只会在通过作用域规则创建类对象或者使用new创建类对象时才会被调用，而并不会在使用malloc()时被调用。同样，析构函数只会在依照作用域规则撤销对象或者使用delete撤销对象时才会被调用，使用free()不会调用析构函数。

如果使用malloc()和free()，应该由客户端代码程序员确保对象有足够的堆内存空间和在不需要对象时还回这个内存。客户代码必须从堆内存中分配空间，并在使用后释放这段堆空间。如果不履行这个职责，会导致内存耗用和内存泄漏。区别类对象的动态管理和对象内存的动态管理是很重要的，后者指的是类数据成员是一个指向动态内存的指针。

程序9-4实现的功能和程序9-3相似，只是使用malloc()取代new来为指针p所动态指向的对象分配空间。显然，这个例子中的内存管理比程序9-3的做法复杂得多。客户代码为未命名的对象分配堆内存空间，然后再为包含该对象的动态内存空间的对象分配动态内存空间（内容为"Smith"）。本程序的输出结果和程序9-3的一样（只是去掉了构造函数和析构函数中的调试信息。）

程序9-4 由客户代码而不是服务器对象进行内存管理

```
#include <iostream>
using namespace std;

struct Name {
     char *contents;                    // public pointer to dynamic memory
  Name (char* name);                    // or Name (char name []);
  void show_name();
  ~Name(); } ;                          // destructor eliminates memory leak

Name::Name(char* name)                  // conversion constructor
```

```
{ int len = strlen(name);              // number of characters
  contents = new char[len+1];          // allocate dynamic memory
  if (contents == NULL)                // 'new' was not successful
    { cout << "Out of memory\n";  exit(1); }    // give up
  strcpy(contents, name); }            // standard set of actions

void Name::show_name()
{ cout <<contents << "\n"; }

Name::~Name()                          // destructor
{ delete contents; }                   // it deletes heap memory, not the pointer

void Client()
{ Name n1("Jones");                    // conversion constructor is called
  Name *p=(Name*)malloc(sizeof(Name));     // no constructor is called
  p->contents = new char[strlen("Smith")+1];     // allocate memory
  if (p->contents == NULL)             // 'new' was not successful
    { cout << "Out of memory\n";  exit(1); }    // give up
  strcpy(p->contents, "Smith");        // 'new' was successful
  n1.show_name();  p->show_name();     // use the objects
  delete p->contents;                  // avoid memory leak
  free (p); // notice the sequence of actions
  }                          // p is deleted, destructor for object n1 is called
int main()              // pushing responsibility to server functions
{ Client();
  return 0;
  }
```

在这个例子中，对象n1是通过作用域规则被自动创建的，其构造函数合理地分配并初始化堆内存。而指针p所指向的未命名对象由malloc()分配空间，没有调用构造函数。调用malloc()只是分配了对象内存，即指针p->contents。而没有为存储该名字信息分配额外的堆内存。因此，客户代码分配并初始化p->contents所指向的堆内存。

当Client()函数终止时，不需要为删除对象n1和还回它所占用的堆内存而操心。作用域规则会撤销该对象，析构函数会释放它占用的堆内存。但指针p所指向的未命名对象就不同了。客户代码不但要撤销该对象，还要还回该对象的堆内存。

在这个例子中，我们使用了类、对象、消息、动态内存管理、构造函数和析构函数，这些都是C++程序设计的重点。然而，这个程序严重违背了面向对象的程序设计思想。当然我们并不是有意违反的。程序员常常在无意中违背了这些设计思想。下面我们再来研究一下这个程序。

首先，违背了数据封装的思想：客户代码直接使用了对象的域contents，这样就增加了模块间的依赖性；如果类Name修改这个域的名字，函数Client()也不得不进行修改。

其次，违背了信息隐藏的思想（从第8章讨论的角度来说）：客户知道类Name使用了堆内存而不是大小固定的字符数组；如果类Name的设计修改了，函数Client()也会受到影响。

这些依赖性增加了程序员间协作开发所需要的信息量。我们需要询问类Name的设计人员许多细节问题，例如域名、动态内存管理、以及其他相关细节等，只是知道其公共数据成员的界面是远远不够的。

函数Client()也并不是由对类Name的成员函数的调用构成。相反，它充斥着大量的数据访问和数据操纵，因此维护人员就要花费额外的时间去理解代码的意图。

最糟糕的是，我们没有把职责推到服务器类，虽然服务器类已经提供了必要的服务，我们在客户代码中进行了内存分配和释放的操作，而不是使用服务器对象来完成。

这样的代码令人沮丧，其复杂性大大超出了我们的想象。而且这样的代码很容易出错，对函数Client()的微小改变就可能把其逻辑联系扯得四分五裂。在下面的例子中，我们先释放p所指向的对象，再试图删除堆内存。结果程序运行时，操作系统提示内存受到破坏，放弃运行。这是合理的结果，因为当p所指向的对象消失时，指针p->contents也消失了。不是每个操作系统都会牺牲运行速度来检查每一个内存访问，在许多平台上这个错误会被忽略。

```
void Client()
{ Name n1("Jones");                                    // conversion constructor is called
  Name *p=(Name*)malloc(unsigned(sizeof(Name)));// no constructor is called
  p->contents = new char[strlen("Smith")+1];           // allocate dynamic memory
  if (p->contents == NULL)                             // 'new' was not successful
    { cout << "Out of memory\n"; exit(1); }            // give up
  strcpy(p->contents, "Smith");                        // 'new' was successful
  n1.show_name(); p->show_name();                      // use the objects
  free (p);                                            // wrong sequence of actions !
  delete p->contents;                                  // there is nothing to delete here!
  }                                  // p is deleted, destructor for object n1 is called
```

此外，如果对象是通过new来分配其空间的，也应该使用delete来释放其空间，如果使用free()将会产生语义错误！同样，用delete来还回用malloc()分配的空间也是语义错误！

在这里使用感叹号的意思是，提醒大家这里的语义错误不同于语法错误、运行时错误或执行的不正确结果（它们可以通过检查测试结果被发现）。对软件工程而言，“语义上不正确的程序”的概念是C++的一个糟糕结果。不正确的调用序列其结果是“没有定义的”。程序员应该确保程序不会包含随时可能造成混乱的代码。

malloc()和free()有两个令人讨厌的特点：不会调用构造函数和析构函数；如果与运算符new和delete混用会造成不正确的程序。这就是C++一般不使用malloc()和free()进行动态内存管理的原因。然而，它们在C程序设计中很常用（C语言没有new和delete），而且遗留的系统也常常使用这些函数。当应用程序要动态地处理大量的内存争夺时，也常常使用这些函数来提高性能。对于精选出来的类，也可以使用函数malloc()和free()来创建自定义的运算符new和delete。运算符的高级使用将在后面讨论。

程序9-3比程序9-4要好得多，因为它没有违背数据封装和信息隐藏，也不会增加额外的程序员协作。它用对服务器对象发送信息来表达算法思路。但是，它加重了客户代码的负担，客户代码需要为指针p所指向的Name对象分配和释放空间。程序员常常在并不十分有用的场合使用动态内存管理。这里就是一个例子。对象应该通过作用域规则而不是使用显式的内存管理来分配和释放空间。

```
void Client()
{ Name n1("Jones");    // conversion constructor is called
  Name n2("Smith");    // no dynamic allocation/deallocation
  n1.show_name();  n2.show_name();
  }                    // destructor for objects n1 and n2 is called
```

总之，应该尽量编写简单的C++程序。

9.4 在客户代码中使用返回的对象

C++函数可以返回基本类型值、指针、引用和对象。函数不能返回数组，但是通过返回指针可以达到与返回数组同样的效果。内部数据类型值只能用作右值，其他返回类型（指针、引用和对象）都能用作左值。这使得C++的源代码语法更富有弹性、更灵活，但同时又会使代码更加难于理解。

在第一次阅读本书时可以跳过本节，虽然这里讨论的程序设计语法很常用。

9.4.1 返回指针和引用

我们从讨论简单的（非复合的）内部数据类型的返回值开始。这种原子类型的返回值可当做右值使用，而指针和引用类型既可作为左值也可以作为右值使用。

我们来考察下面的类Point。其成员函数setPoint()修改目标对象Point的状态；成员函数getX()和getY()返回整数值；成员函数getPtr()用于返回指向数据成员x的指针；成员函数getRef()返回指向数据成员x的引用。这里没有提供返回指向数据成员y的指针和引用的成员函数，因为函数getPtr()和getRef()已经足够演示相关的论题，包括修改对象的状态。

```
class Point
{ int x, y;                          // private data
  public:
    void setPoint(int a, int b)
    { x = a;  y = b; }
  int getX()                         // return a value
    { return x; }
  int getY()
    {return y; }
  int* getPtr()                      // return a pointer to the value
    { return &x; }                   // the address operator is needed
  int& getRef()                      // return a reference to a value
    { return x; } } ;                // no address operator for reference
```

为了弄清楚是否应该使用取地址运算符，我们采用赋值和参数传递时使用的相同逻辑。函数getPtr()返回一个指针，所以直接返回x的值会造成类型不匹配，即语法错误。函数getRef()返回引用，而且该引用可以（也应该）被它即将指向的数值初始化，所以使用&x将会产生类型不匹配，即语法错误——&x是一个地址而不是一个int值。

当一个值从函数返回时，它只能当做右值使用，即只能出现在赋值等号的右边，或者出现在比较语句中（或者作为函数调用的输入参数）。在下面的例子中，客户代码操纵从函数返回的值。这个值改变了，但是从函数返回的对象值没有改变，因为返回的值是原来值的副本，就像按值传递参数一样。

```
Point pt;  pt.setPoint(20,40);
int a = pt.getX(), b = 2* pt.getY() + 4;        // ok, use as rvalue
a += 10;                              // 'a' changes, but pt.x does not
```

当从函数返回一个指针或者引用时，它既可以当做右值使用，又可以当成左值使用，既可以出现在赋值等号的左边，也可以作为函数调用的输出参数。在接下来的例子中，第一行代码把getPtr()和getRef()的返回值当作左值来使用，这是合法的。第二行代码修改

了指针ptr和引用ref指向的值。注意两者都是指向变量pt的内部数据成员pt.x。这个数据成员是私有的，但客户代码可以不使用访问函数而修改它。第三行代码使用函数调用作为左值，也修改了变量pt的状态。注意在表达式 *pt.getPtr()中不需要使用括号，因为点选择运算符“.”的运算优先级高于间接引用运算符的运算优先级，所以这里表示的是对由该方法返回的值进行间接引用，而不是表示一个指向目标对象的指针。(pt不是指针，而是一个Point对象的名字。)

```
int *ptr = pt.getPtr(); int &ref = pt.getRef();    // ok, use as rvalue
*ptr += 10; ref += 10;  // private data is changed through aliasing
*pt.getPtr()=50; pt.getRef()=100;          // private data is changed
```

首先，把函数调用当成左值使用的语法是不常见的。第二，有些人认为这种使用方法“破坏了数据封装和信息隐藏”，它改变了客户代码不能访问的私有数据的值。但是，谁说信息隐藏指的是不能改变私有数据的值呢？调用setPoint()就改变了私有数据，而这并没有违背信息隐藏，getRef()亦是如此。数据封装和信息隐藏指的是避免类之间的依赖性，而不是说不能改变私有的数据成员的值。

从软件工程的角度看，本示例的主要问题是使用了变量别名——数据成员x可以用很多其他名字代替，如ptr、ref、getPtr()和getRef()等。这些名字，特别是getPtr()和getRef()根本无法从字面上推断出它们指向数据成员x。因此这种编码语法强迫维护人员投入额外的精力去理解代码的意义。因此使用这种技巧一定要小心，它是合法的C++语句，但是很危险，甚至比使用全局变量还要有害。

返回指针和引用要求传递给调用者的地址在函数终止后保持有效。在前面的例子中，getPtr()和getRef()返回指向pt.x的指针，而pt.x在函数调用返回后仍然在有效的作用域中。但有时候并非如此。下面的例子中，函数getDistPtr()和getDistRef()计算目标Point对象和原点间的距离，它们返回指向计算出来的距离值的指针和引用。这就犯了令人遗憾的大错误。

```
class Point
{ int x, y;
public:
  . . .              // setPoint(), getX(), getY(), getPtr(), getRef()
  int* getDistPtr()
  { int dist = (int)sqrt(x*x + y*y);
    return &dist; }          // no copying, but dist disappears
  int& getDistRef()
  { int dist = (int)sqrt(x*x + y*y);
    return dist; }  } ;      // different syntax, same problem
```

局部变量dist在函数getDistPtr()和getDistRef()终止后消失。如果它占用的空间没有被其他变量占用，使用它的地址可能会产生正确的结果，否则可能会没有任何警告地产生不正确的计算结果。一些编译程序可能会产生警告，但是另一些编译程序则不会。不管怎样，上面那个版本的Point代码和下面的客户代码都是语法上的错误。

```
Point pt;  pt.setPoint(20,40);
int * ptr = pt.getDistPtr();                              // invalid pointer
cout << " Pointer to distance : " << *ptr << endl;        // okay
int &ref = pt.getDistRef();                               // invalid reference
cout << " Reference to distance : " << ref << endl;       // okay
```

```
cout << " Pointer to distance : " << *ptr << endl;      // bad
cout << " Reference to distance : " << ref << endl;     // bad
```

程序在我们的计算机上的运行结果如图9-5所示。从图中可以看到，无效指针和无效引用的第一次使用结果欺骗了我们——虽然指针和引用都是无效的，但输出值44是正确的。但再次试图打印这些值就产生了错误的结果。这意味着，以后再使用这些值就不正确了。但这很可能会被人忽视。既然我们已经检查过ref和*ptr的值，并且看到了正确的结果，我们没理由想到它们会在以后被修改！这样程序员的警觉性转移到了其他方面。



图9-5　返回指针和引用的正确和错误结果

通常，我们认为如果程序的结果是正确的，那么程序自然也是正确的。但是，我们应该再选一组数据，通常这组数据会覆盖程序的其他路径，它的测试结果也应该是一样的。

警告　当函数返回一个指针或者引用时，我们必须确保它指向的位置不会被C++作用域规则设置为无效的。违反这个指导思想不是语法错误，但是运行结果正确就不能证明我们的程序也是正确的。

一般来说，比较好的做法是：限制函数的返回值必须为布尔标志，以告诉客户代码函数调用的成功与失败。然而使用诸如getX()、getY()一类的函数的优越性又是如此明显，程序员通常会选择使用这样的函数。一定不要因为C++的强功能特性过于兴奋，也不要让函数返回指针和引用，特别是不要返回指向马上就会无效的值的指针和引用。编译程序不能防止我们犯这样的错误。

9.4.2　返回对象

在接下来的例子中，我们将为类Point再增加3个成员函数：closestPointVal()、closestPointPtr()和closestPointRef()。每个成员函数都接收一个指向Point对象的引用作为参数，并计算这个参数到原点的距离，以及消息的目标对象到原点的距离。如果参数离原点比较近，函数返回参数对象；如果消息的目标对象离原点比较近，函数就返回目标对象（即对指向目标对象的指针this间接引用）。

第一个函数返回最近的对象本身，第二个函数返回指向最近的对象的指针，第三个函数返回指向最近的对象的引用。函数接口的优点是可以用于链式消息表达式，即一个函数调用的返回值可以做为另一个函数调用的目标。有三种类型的返回值既可以当作左值也可以当做右值使用，它们是对象、指针和引用。

```
class Point
{ int x, y;
public:
  . . .                  // setPoint(), getX(), getY(), getPtr(), getRef()
  . . .                           // getDistPtr(), getDistRef()
  Point closestPointVal(Point& pt)
```

```
{ if (x*x + y*y < pt.x *pt.x + pt.y * pt.y)
    return *this;            // object value: copying to temp object
  else
    return pt; }             // object value: copying to temp object
Point* closestPointPtr(Point& p)        // returns pointer: no copy
{ return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p; }
Point& closestPointRef(Point& p)        // returns reference: no copy
{ return (x*x + y*y < p.x*p.x + p.y*p.y) ? *this : p; } } ;
```

this是一个关键字，表示指向消息的目标对象（即当前对象）的指针，如下面的代码中this表示指向对象p1的指针。第一个函数使用了长表达形式（两个return语句），后两个函数使用了缩写形式（使用了条件运算符）

注意取地址方式在返回对象值时所发挥的作用。函数closestPointVal()（按值）返回一个Point对象，如果返回的是目标对象，（指向目标对象的）this指针必须先被间接引用，接着目标对象（对象p1）的域值被复制到接收对象（对象pt）的域中。如果返回的是参数对象，则使用引用p。这个引用是它所指向的对象（对象p2）的同义词，该对象的所有域被复制到接收对象（对象pt）中。

```
Point p1,p2;  p1.setPoint(20,40);  p2.setPoint(30,50);     // set Point objects
Point pt = p1.closestPointVal(p2);     // fields of the closest point are copied
```

函数closestPointPtr()返回指向离原点最近的Point对象的指针。当目标对象比参数对象更接近原点时，就返回this指针（它指向目标对象，例如p1）。在下面的例子中，指针值被复制到接收指针（指针p）中。当参数对象更接近原点时，使用引用p。因为这个引用是其指向的对象的同义词（而不是该对象的地址），所以&p的值被复制到接收指针中。这个指针可以用来访问最近的对象（p1或者p2）的成员。

```
Point *p = p1.closestPointPtr(p2);      // pointer is returned: fast
p->setPoint(0,0);              // move p1 or p2 to the point of origin
```

我们可以看到，返回对象值很慢，而返回对象指针可以避免复制对象域。而返回引用的情况就不那么明显了。函数closestPointRef()返回指向最近的Point对象的引用。如果目标对象更接近原点，应该使用this指针。既然不能把一个指针赋值给引用变量，我们就应该为目标值使用*this这个表示方法。但是，要记住，这并不意味着创建了目标对象的副本。只是使用了这种表示方法而已。与按引用传递参数类似，只是复制了地址（引用），而没有复制对象的域。如果参数对象更接近原点，应该使用引用p。该操作的结果也一样：只复制了引用，没有复制域。

```
Point &r =p1.closestPointRef(p2);     // reference is returned: fast
r.setPoint(0,0);              // move p1 or p2 to the point of origin
```

但是，如果客户代码中的接收变量是这种对象类型，而不是引用类型，仍然会发生复制操作

```
Point pt = p1.closestPointRef(p2);     // p1 or p2 is copied into pt
```

所以返回引用类型并不一定能消除潜在的性能问题。

返回对象（无论是按值返回或者按指针、按引用返回）的主要好处是，可以使用消息的链式符号：即向由函数返回的对象发送消息。

```
Point p1, p2;  p1.setPoint(20,40);  p2.setPoint(30,50);
```

```
int a = p1.closestPointVal(p2).getX();          // might be slow
int b = (*p1.closestPointPtr(p2)).getX();       // fast and elegant
int c = p1.closestPointRef(p2).getX();          // fast and elegant
```

上面代码中closestPointVal()返回的对象是一个临时的未命名的Point对象，它一直存在直到接收完getX()消息，然后该无名对象消失。在其他两个函数调用中，指针和引用指向的都是在客户空间中定义的对象，因此没有目标对象生存期的问题，它们一直存在着。

在上面的例子中，消息发送一个返回的对象，并没有改变这个对象的状态。链式符号也可能使用改变目标对象状态的消息。

```
p1.closestPointRef(p2).setPoint(15,35);   // what is set here? p1? p2?
p1.closestPointPtr(p2)->setPoint(10,30);  // and what is set here?
```

在上面的例子中，客户代码改变了对象p1或者p2的值，而且修改是持续有效的。而在下面的例子中，被改变的是一个临时的未命名对象，该对象在被改变后立刻被撤销。因此这样的代码是没有什么意义的，但在C++中是合法的。

```
p1.closestPointVal(p2).setPoint(0,0);   // create, set, destroy object
```

使用返回的对象一定要小心，在性能上获得的提高和表达方法上的方便性常常不值得牺牲完整性和面临操作结果带来的迷惑性。而且，创建和撤销未命名对象也消耗时间，因为相关的堆内存管理、构造函数与析构函数的调用都需要时间。

9.5 关于const关键字的讨论

这部分内容很重要。我们将会复习关键字const的多种含义，并学习如果将这个关键字用于软件开发者的最重要任务之一——将开发人员关于程序组件的知识传达给维护人员。如果不能很好地完成这个任务，很容易（通常）就会导致软件问题。

正如我们在前面（第4章和第7章中）所看到的那样，关键字const在C++中有多个含义，其意义依赖于所在的上下文。如果把const放在变量的类型名前，则说明此变量的值是保持不变的。该变量必须在定义时初始化，而且任何将它赋值为其他值（甚至是同样的值）的企图都会标记为语法错误。

```
const int x = 5;     // x will not (and cannot) change
x = 20;              // syntax error: it prevents changes to x
int *y = &x;         // syntax error: it prevents future changes to x
```

当指针指向一个常量变量时（这个说法值得注意，虽然称为变量，但是不能修改），必须在类型名前使用const关键字以标识这个指针为常量。这样，任何使用被间接引用的指针作为左值的企图都会标记为语法错误。

```
const int *p1 = &x;  // ok: *p1 will not be used to change x
*p1 = 0;             // syntax error: *p1 cannot be an lvalue
int a = 5;           // an ordinary variable: it can be changed
p1 = &a;   *p1 = 0;  // syntax error: 'a' cannot change through *p1
```

当引用指向一个常量变量时，也必须在类型名前使用const关键字以标识这个引用为常量。然后，任何使用这个引用作为左值的企图都会标记为语法错误。

```
int &r1 = x;                // syntax error: x should not change through r1
const int &r2 = x;          // ok: reference to a constant, x will not change
r2 = 0;                     // syntax error: r2 is a reference to a constant
```

```
const int &r3 = a;              // 'a' can change but not through r3
r3 = 0;                         // syntax error: 'a' cannot change through r3
```

当指针运算符后面紧跟关键字const时，说明这是个指针常量：即它指向相同的位置，而不能转为指向另一个位置。但是并不保证该指针指向的值是保持不变的。

```
int* const p2 = &a;             // p2 will point to 'a' only, not elsewhere
*p2 = 0;                        // ok: no promises were made to keep it const
int b = 5;  p2 = &b;            // syntax error: breach of promise
```

不需要使用特别的表示方法说明引用是常量。C++中所有的引用缺省为常量，不能转而指向另一个位置。和指针一样，也不能保证该引用指向的值是保持不变的。

```
int& r4 = a;            // r4 points to 'a' only, no const is needed for pledge
r4 = b;                 // no syntax error; just r is not diverted
```

const在函数接口中的使用和在值、指针中的使用相似：它表示实际参数（或指针）不能作为调用的结果被修改。

```
void f1(const int& x);      // x is not changed by the function
void f2(const int x);       // redundant: x is passed by value anyway
void f3(int* const y);      // redundant: y is passed by value
void f4(int * const *y);    // ok, pointer is passed by pointer
void f4(const int *&y);     // ok: pointer is passed by reference
```

在函数接口中使用const，就更难让函数返回指向这个参数对象的指针或者引用。如果允许的话，客户代码就有可能通过别名修改常量，其情况与本章前面讨论的例子相似。

如果大家跳过了前一部分没有读，那么这里再次重复说明三个实现相同功能的函数：closestPointVal()、closestPointPtr()和closestPointRef()。每个函数都将目标对象到原点之间的距离和参数对象到原点之间的距离进行比较。如果目标对象距离原点更近，每个函数都返回目标对象；如果参数对象距离原点更近，每个函数返回参数对象不同的地方在于，closestPointVal()返回对象本身，closestPointPtr()返回指向对象的指针，closestPointRef()返回对象的引用。9.4节“在客户代码中使用返回的对象”中，我们没有在函数参数前使用const关键字，在下面的例子中我们添加这个关键字。

```
class Point
{ int x, y;                                     // private data
public:                                         // public operations
  . . .                                 // setPoint(), getX(), getY(), getPtr(), getRef()
  . . .                                         // getDistPtr(), getDistRef()
  Point closestPointVal(const Point& pt)  // irrelevant: data is copied
  { if (x*x + y*y < pt.x *pt.x + pt.y * pt.y)
      return *this;                             // object value: copying to temp object
    else
      return pt; }                              // object value: copying to temp object
  Point* closestPointPtr(const Point& p)  // parameter is const
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p; }         // error
  Point& closestPointRef(const Point& p)  // parameter is const
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? *this : p; } };      // error
```

函数closestPointVal()返回的或者是目标对象或者是参数对象，但是不管是哪一个都是Point对象的副本。因此，函数参数的const关键字并没有限制函数的使用。如果在客户代码修改了返回的对象，这种改变只会影响实际参数对象的副本而不会影响保证不变的

实际参数对象。

```
Point p1,p2;
p1.setPoint(20,40);  p2.setPoint(30,50);  // set Point objects
Point pt = p1.closestPointVal(p2);
pt.setPoint(0,0);                         // no breach of pledge
p1.closestPointVal(p2).setPoint(0,0); // not useful, and not harmful
```

函数closestPointPtr()可以返回指向其Point实际参数的指针。客户代码就可以使用这个指针修改实际参数对象的状态。同样，函数closestPointRef()可以返回其Point实际参数的引用。这个引用也可以用来修改实际参数对象的状态。

```
Point *p = p1.closestPointPtr(p2);       // p2 should not be changed
p->setPoint(0,0);         // p2 could be changed - breach of promise
Point &r =p1.closestPointRef(p2);        // p2 should not be changed
r.setPoint(10,10);        // p2 could be changed - breach of promise
```

在这个例子中，如果程序能运行，对象p2的值也不会真正改变，因为所有的三个函数都是返回比对象p2距离原点更近的对象p1。即使修改了对象p2，它也是在函数closestPointPtr()和closestPointRef()外部被修改的！不过不用担心，C++不允许这样使用常量对象。但是，编译程序在分析客户代码时很难发现这种错误的使用（对人们来说也很难发现），因此它会将两个函数都声明为错误。

声明为错误的形式化解释是，参数对象（例如closest PointPtr ()函数中的参数对象）使用了const关键字，但是返回类型没有使用。

```
Point* closestPointPtr(const Point& p)        // inconsistency: damage to const
{ return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p; }        // syntax error
```

C++提供了3个解决方法处理这种情况。第一个方法是从参数界面中放弃使用const关键字。第二个方法是在成员函数内部，使用const_cast运算符移去const属性。第三个方法是在另外两个合适的位置增加const关键字。

从参数界面中删除const关键字只适用于那些十分失望和担心的程序员。一个真正的程序员是不会放过任何机会将设计人员在设计类时的思路传达给客户代码的程序员和维护人员的。该函数没有修改其参数对象，因此应该使用const关键字。

第二个办法较为复杂。const_cast运算符强行将常量参数转换为相同类型的变量，但是去除了不能被修改的保护。在const_cast运算符和参数的尖括号中指定类型。例如const_cast<valueType>(constValue)把一个valueType类型的变量constValue强制转换到相同的valueType类型的值，并移去不能被修改的保护。对于类Point，我们可以使用const_cast<Point*>(&p)，将指向常量Point对象的指针转换为指向非常量Point对象的指针。

下面这个版本的Point类在参数值从成员函数中返回时，移去了参数的常量属性。

```
class Point
{ int x, y;
public:
  . . .                          // setPoint(), getX(), getY(), getPtr(), getRef()
  . . .                            // getDistPtr(), getDistRef(),closestPointVal();
  Point* closestPointPtr(const Point& p)                  // prevents damage to p
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : const_cast<Point*>(&p); }
```

```
  Point& closestPointRef(const Point& p)                    // prevents damage to p
{ return (x*x+y*y < p.x*p.x+p.y*p.y) ? *this : const_cast<Point&>(p);
  } } ;
```

现在，修改返回对象的客户代码就合法了。但这是一个粗略的解决方法，我们之所以要在这里介绍是为了较全面地讲述C++的知识，而不推荐使用它。相比较而言，这种做法比移去参数界面中的const关键字要好一点，因为删除const关键字也去除了函数内部对参数使用的保护。而使用const_cast只是在特定的操作中（本例中是指返回值的操作中）去除了保护，而不是在所有地方都去除保护。但是const_cast这种方法很糟糕，不容易理解。

让closestPointPtr()和closestPointRef()通过编译的最好方法是，通过让它们返回常量来保证不修改返回的对象，即在函数返回值前加上const关键字。这就是const关键字的第三种含义（第四种含义会在稍后介绍）。在函数返回值前加上const关键字，就可以防止函数调用者修改函数的返回值。也就是说，返回值只能当做右值使用而不能当做左值使用。

```
class Point
{ int x, y;
public:
// . . .  setPoint(), getX(), getY() and so on
const Point* closestPointPtr(const Point& p)
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p; }   // okay
const Point& closestPointRef(const Point& p)
  { return (x*x+y*y < p.x*p.x+p.y*p.y) ? *this : p; } ;   // okay
```

这样，客户代码使用Point对象就有更严格的限制了。

```
Point p1,p2;  p1.setPoint(20,40);  p2.setPoint(30,50);
Point *ptr = p1.closestPointPtr(p2);          // syntax error: should be const
Point &ref = p1.closestPointRef(p2);          // syntax error: should be const
const Point *p = p1.closestPointPtr(p2);      // *p is an rvalue
p->setPoint(0,0);                             // syntax error: no change to object
const Point &r =p1.closestPointRef(p2);       // r cannot be an lvalue
r.setPoint(10,10);                            // syntax error: no change to object
```

那么，使用指针p和引用r的好处是什么呢？答案是很明显的，它们不能调用诸如setPoint()一类的函数修改目标对象，但是可以调用诸如getX()一类的不修改目标对象的函数，像下面这样。

```
int x1 = p->getX();              // p points to a constant Point
int x2 = r.getX();               // r refers to a constant Point
```

如果喜欢使用链式符号，我们可以用下面这种方法在指针和引用后面添加方法调用，以获得最近的点的坐标。

```
x1=(p1.closestPointPtr(p2)).getX();     // syntax error
x2=p1.closestPointRef(p2).getX();       // syntax error
```

即使我们对语法的细节还不够熟悉，也希望可以领会这些讨论的一般性思路。C++提供关键字const是为了方便编译程序和维护人员弄清楚一个实体在执行过程中是否被修改。我们讨论的正是防止修改数值、指针、函数参数、参数指针的方法，以及防止修改返回一个指针或一个引用的函数的返回值的方法。

下面再让我们来考察上面的代码。一个指向常量对象的指针或者引用不能用来调用像

setPoint()这样的函数，这是很自然的，因为setPoint()修改了指针或者引用所指向的对象的状态。但使用getX()有什么错误呢？这个函数不会修改指针和引用所指向的对象的状态——或者，也许该函数可以修改？

这又是我们在第7章讨论过的与函数参数相关的基本意识形态问题。我们怎么知道函数修改了它的参数还是保持它不变呢？我们并不想去研究函数的代码，而只是根据函数头来判断。如果函数头声明参数为const，我们就知道该参数不会改变；如果函数头没有声明参数为const，我们就认为该参数被修改了，而不管函数实际上是否修改了该参数。还记得那个借喻吗？考古学家认为，如果有铜轴电缆可以很好地证明古代使用了电话，那么没有铜轴电缆就可以很好地证明古代使用了蜂窝电话。

C++编译程序就遵循相同的逻辑。它只是在函数头中查找const关键字，并将对该参数的修改标记为语法错误。但是它并不够聪明，不知道遍历函数代码以独立地判断参数是否被修改了。它武断地认为如果没有const就表明参数会被修改。

我们再来看函数setPoint()和getX()。我们是如何知道前一个函数修改对象而后一个函数不会修改呢？这是因为我们看过了函数代码，而且我们相信函数名，这是很明显的，不是吗？但是编译程序并不知道这些。编译程序将对setPoint()的调用标记为语法错误，不是因为它知道setPoint()修改了对象，而是因为它没找到证据证明setPoint()没有修改对象。对编译程序而言，getX()和setPoint()都是同一类型的。如果没有证据证明getX()保持对象不变，编译程序就会认为getX()修改了对象的状态。

这里就是C++以第四种含义使用const的场合：把关键字插到函数列表的闭圆括号“)”与函数体的开花括号之间。在原型中，关键字被插在参数列表的闭圆括号与分号之间。下面的类Point显式地表明其成员函数对以下三实体进行了什么操作：a) 函数参数，b) 函数的返回值，c) 目标对象的数据成员。

```
class Point
{ int x, y;
public:
  void setPoint(int a,int b)   // it modifies fields, right?
    { x = a;  y = b; }
  int getX() const    // it does not modify fields: see the evidence?
    { return x; }
  int getY() const    // it does not modify fields: see the evidence?
    { return y; }
const Point& closestPointRef(const Point& p) const // isn't it nice?
  { return (x*x+y*y < p.x*p.x+p.y*p.y) ? *this : p; } } ;
```

非常棒，不是吗？这里的讨论的确很复杂，但是C++中的关键字const确实有这么多的含义！在下一节中我们至少还要再讨论一种其他的含义。一定要认真对待这个关键字。在编写服务器代码时，这个关键字是将我们在设计时的思想传达给其他人的主要工具。在阅读代码时，该关键字也是我们理解设计人员意图的主要工具。要在任何可能的地方使用const关键字，如果我们不将对类成员函数的知识传达给客户端代码程序员和维护人员，就是一个很严重的错误。

提示　使用const关键字以表明一个值（或者指针）在初始化后就不再被修改；使用该关键字以表明在函数执行过程中不会改变函数参数（或者指针）；使用该关键字以表明客户代码不会修改函数返回的值（按指针或引用传递）；使用该关键字以表

明成员函数不会在消息调用中修改目标对象的状态。在学习C++代码时，也要研究const关键字的适用方法，不要大意。

9.6 静态类成员

在这一部分里，我们将对类数据成员的表示符号进行概括。从概念上来说，一个类是对象的蓝图。类规格说明描述了这个类的每一个对象拥有的数据和函数

这就是为什么每建立一个类对象实例，都会为该对象实例创建一个单独的数据成员集合无论我们使用何种方式创建对象，通过将对象定义为有名的局部或者全局变量来创建，或者使用new运算符将对象定义为未命名的动态变量来创建，还是按值传递一个对象作为函数的参数来创建，或者从一个函数按值返回一个对象来创建，无论如何，每个对象实例都有自己private、public、protected数据成员值集合。

没有必要为每个对象创建单独的成员函数集合。每个成员函数的目标代码都只产生一次除了由程序员指定的参数外，每个程序函数还有一个隐式的参数，即指向目标对象的指针当将每一特定的对象作为成员函数的目标时，这个指向目标对象的this指针就被传递给函数。该函数就可以对目标对象的数据成员进行操作。

9.6.1 用全局变量作为类特性

有时候，为一个类的所有对象提供共有的数据成员副本，比在每个类对象中维护单独的副本可以更加有效合理地利用内存。

例如，应用程序可能需要对类对象实例计数，如包含数据成员count的类Point。从逻辑上来说，这个数据成员和其他数据成员一样都从属于类。

```
class Point {
    int x, y;              // individual for each Point object
    int count;             // common for all Point objects
  . . . } ;
```

事实上，这个程序有很多问题。我们只需要一个count，如果应用程序创建了1 000个Point对象，定位这1 000个count数据域并在每个数据域中保持相同的值简直是毫无意义的而且，如何去维护这个域呢？每创建一个新的Point对象，我们都得把count加1。这意味着在Point的构造函数执行这个操作比较好。类似地，在析构函数中将对象的count域减1比较合适。

```
Point::Point (int a, int b)        // general constructor
  { x = a;  y = b;  count++; }     // increment the count of objects
```

这个解决方法并不好，它只是递增了新创建的对象中的一个count值，而没有递增其他对象的数据成员。而且，正在被创建的对象的count数据域也没有被初始化为早先创建的对象的count数据域的值。因此，这个构造函数将一个未经初始化的值加1，是行不通的。

使用全局变量可以解决这个问题。全局变量可以在整个程序开始运行前初始化为零，然后，在创建新对象时把该全局变量加1（在构造函数中），在撤销一个对象时把这个值减1（在析构函数中）。

例如，可以使用一个全局变量对被实例化的Point个数进行计数，在构造函数中递增这

个计数，在析构函数中递减这个计数。程序9-5就显示了使用这个方法的类Point的实现。Point的构造函数Point既可以用作缺省构造函数（客户代码不提供参数），也可以当成转换构造函数（客户代码提供一个参数），还可以作为一般的构造函数（客户代码提供表示点坐标的两个参数）。为了进行演示，构造函数和析构函数里都添加了语句用来跟踪函数调用的顺序。函数quantity()返回count的值，这样即使全局变量的名字改变了，客户代码也不需要修改。全局变量count被显式地初始化为0。根据C++的语言规则，也可以隐式地初始化为0，但是显式的初始化比较好。

程序9-5　使用全局变量对对象实例计数

```
#include <iostream>
using namespace std;

int count = 0;          // does maintainer know it belongs to Point?

class Point {
     int x, y;                        // private coordinates
  public:
     Point (int a=0, int b=0)         // general constructor
       { x = a;  y = b;  count++;
         cout << " Point created: x=" << x << " y=" << y << endl; }
     void set (int a, int b)          // modifier function
       { x = a;  y = b; }
     void get (int& a, int& b) const       // selector function
       { a = x;  b = y; }
     void move (int a, int b)         // modifier function
       { x += a;  y += b; }
     ~Point()                         // destructor
       { count--;
         cout << " Point destroyed: x=" << x << " y=" << y << endl; }
 } ;

int quantity()                        // access to global variable
  { return count; }

int main()
{ cout << "Number of points: " << quantity() << endl;
  Point *p = new Point(80,90);        // dynamically allocated object
  Point p1, p2(30), p3(50,70);        // origin, x-axis, general point
  cout << "Number of points: " << quantity() << endl;
  return 0;                  // dynamic object is not properly deleted
  }
```

程序的运行结果如图9-6所示。可以看到，我们先创建了一个未命名的Point对象，接着使用缺省构造函数建立了第一个命名指针（p1），再使用转换构造函数创建下一个有名对象（p2），最后使用一般构造函数建立对象p3。这些命名的Point变量按逆序被撤销。注意，我们没有合理地删除未命名的动态Point对象，因此不会看到表示它被撤销的输出信息。

这种解决方法是可行的，但是将它用在大型程序中会有几个缺点。首先是在客户代码中的任意位置都可以修改变量count的值，这将会增加模块间的依赖性。其次，全局变量名可能和工程项目中的其他全局名或者函数库中的命名发生冲突。这样，即使大部分的开发组成员不必了解类Point，每个开发组成员也需要注意count这个变量名。这就增加了程序员对

功能的其他部分需要了解的信息量，对开发人员和维护人员而言，工程的复杂性都增加了。

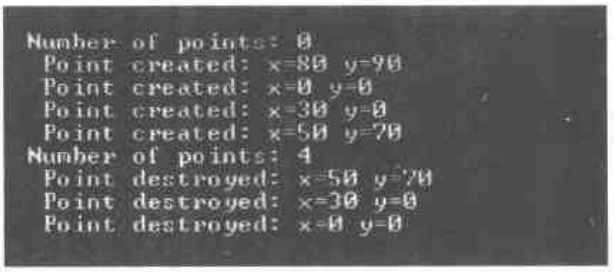


图9-6 程序9-5的输出结果

然而，这个解决方法的最大的问题是没有向维护人员传达设计人员在设计时的思路。在定义全局变量count时，我们知道这个变量用来计算Point对象的数目，而不是计算Rectangle或者其他对象的个数。但语法上没有任何表示说明这个变量和特定的类相关。维护人员不得不阅读注释（而注释可能很难理解、很晦涩或者根本没有）或者研究大量的源代码，才能分析出这一点。

9.6.2 关键字static的第四种含义

在C++中，我们可以通过重用关键字static来解决这个问题。在第6章，我们已经学习了static的三种含义。

使用static的第一种含义是表明一个全局变量只对定义在同一文件中的函数可见，即使在另一个文件中使用extern，另一个文件中的函数也不能访问它。

在第二种含义中，关键字static用于函数的局部变量。它表明该变量的值不会因为函数的终止而丢失（其他的局部变量值会丢失），而是被系统保存下来，如果再次调用该函数，这个保存下来的值会用来初始化该变量。在第三种含义中，关键字用于修饰函数，表明该函数只能在同一文件中调用。

在这一部分里，我们将把static用于类的数据成员。其含义正是我们所需要的——它表明对类的所有对象这个数据成员都只有一个实例。该实例是类类型的所有对象共有的。

从其他方面看，静态数据成员也是普通的数据成员，也可以被声明为公共的、受保护的或者私有的。定义和访问静态数据成员的语法也和其他的类数据成员的语法相同。惟一的不同是使用了关键字static。

```
class Point {
     int x, y;                          // private coordinates
     static int count;                  // another meaning of the keyword
  public:
     Point (int a=0, int b=0)           // versatile constructor
       { x = a;  y = b;  count++; }
     void set (int a, int b)            // modifier function
       { x = a;  y = b; }
     void get (int& a, int& b) const    // selector function
       { a = x;  b = y; }
     void move (int a, int b)           // modifier function
       { x += a;  y += b; }
     ~Point()                           // destructor
       { count--;  } } ;
```

这里的数据成员count是所有的Point类对象实例都可以访问的共享值。无论类定义是否在当前（可访问的）作用域中，该数据成员都是可访问的。

9.6.3 静态数据成员的初始化

和非数据成员的全局变量相似，静态成员数据在类规格说明外部被初始化。但和全局变量count不同的是，静态数据成员count是一个类数据成员，它必须被显式地初始化——不可以对静态或者非静态的数据成员进行隐式初始化。和在类括号外部使用任何类成员名一样，也应该使用类作用域运算符来表明这个数据成员所属的类。

```
int Point::count  = 0;  // this is not an assignment (see the type name here?)
```

在前面我们多次提到C++中赋值和初始化的区别。赋值运算符通常可以表示赋值或者初始化。如果在类型名后紧接着变量名，则表示是初始化；如果变量名前没有类型名，则表示是赋值。在这里，初始化和赋值之间的区别是很重要的。初始化对公共和非公共的静态数据成员是合法的，其语法格式也和一般的初始化一样。但对非公共的静态数据成员进行赋值是不合法的。

```
Point::count  = 0;       // assignment (illegal for private 'count')
```

静态数据成员只能被初始化一次。所以，这条初始化语句应该和其他的成员函数的定义一起放在类的实现文件.cpp中，而不能放在类的头文件中。

访问静态数据成员和访问非静态数据成员一样。非成员函数（例如quantity()）不能直接访问私有静态数据成员。为了避免这个问题，我们可以把函数quantity()编写为成员函数。

```
class Point {
    int x, y;                          // private coordinates
    static int count;                  // another meaning of keyword
  public:
  Point (int a=0, int b=0)             // versatile constructor
    { x = a;  y = b;  count++; }
    int quantity() const               // no change to object state
    { return count; }
  . . . } ;
```

注意到前一个版本的quantity()函数（全局函数）并没有使用const修饰符。因为只有成员函数才可以保证不修改目标对象的数据成员，而全局函数没有目标对象。

一个静态数据成员不能是联合的成员，也不能是位域的类。联合和位域都表示了属于某个特定对象的特殊内存用途。静态数据成员不属于某个特定对象，而是属于整个类。因为我们可能并不经常使用联合域和位域，因此这个限制对我们来说没有什么影响。

9.6.4 静态成员函数

我们已经基本上掌握了类的相关知识，但在本章结束之前，再介绍一下static在C++中的第5种含义。这个关键字也可以用来修饰不访问非静态数据成员的类成员函数。这意味着一个静态成员函数只能访问它的参数、类的静态数据成员和全局变量。

对于类Point来说，quantity()函数最有资格成为一个静态成员函数。它没有任何

参数，只访问静态数据成员count，除此之外没有访问任何非静态数据成员。

```
class Point {
     int x, y;                          // private coordinates
     static int count;                  // private count of objects
  public:
     Point (int a=0, int b=0)           // versatile constructor
     { x = a;  y = b;  count++; }
     static int quantity()              // it cannot be const
     { return count; }
  . . . } ;
```

静态成员函数不能被声明为const，即使它没有改变所访问的任何值。一个静态成员函数访问的值是其参数、静态数据成员和全局变量，而这些数据都不是对象状态的一部分。对成员函数中使用关键字const是表明，函数不会修改该函数访问的目标对象的数据成员。既然一个静态成员函数根本不访问非静态数据成员，也就没有必要使用const。

这样的解释可能不够说服力，但是实际上有个更加合乎逻辑的解释。我们已经演示了一系列的示例，首先实现的函数quantity()是一个全局非成员函数，它访问全局变量count。后来我们将全局的count变成静态类数据成员，quantity()函数也随之成为类Point的成员函数。作为非静态成员函数，它被定义为const，以说明没有修改非静态的类数据成员。最后，我们把这个函数变成静态成员函数，const就显得没什么必要了。

与静态数据成员相似，静态成员函数可以通过目标对象（或者指向类对象的指针）来调用，这和非静态成员函数的调用是一样的。另外，即使没有创建类对象，也可以使用类作用域运算符直接调用它：

```
int main()
{ cout << "\nNumber of points " << Point::quantity();   // it prints 0
  Point p1(20,40);
  cout << "\nNumber of points " << p1.quantity();      //  it prints 1
  cout << "\nNumber of points " << Point::quantity();  // it prints 1
... }
```

程序9-6演示的类Point使用了静态数据成员count。虽然count是私有的，但它在类定义外初始化。函数quantity()也被定义为静态成员函数，因此可以使用类作用域运算符（第一个调用）和目标对象来访问（第二个调用）。

程序9-6 使用静态数据成员和成员函数

```
#include <iostream>
using namespace std;


class Point {
     int x, y;                          // private coordinates
     static int count;
  public:
     Point (int a=0, int b=0)            // general constructor
       { x = a;  y = b;  count++;
         cout << " Point created: x=" << x << " y=" << y << endl; }
       static int quantity()            // const is not allowed
            { return count; }
       void set (int a, int b)          // modifier function
```

```
        { x = a;  y = b; }
        void get (int& a, int& b) const   // selector function
        { a = x;  b = y; }
        void move (int a, int b)          // modifier function
        { x += a;  y += b; }
        ~Point()                          // destructor
        { count--;
          cout << " Point destroyed: x=" << x << " y=" << y << endl; }
 } ;

int Point::count = 0;

int main()
{ cout << " Number of points: " << Point::quantity() << endl;
  Point p1, p2(30), p3(50,70);            // point of origin, x-axis, general point
  cout << " Number of points: " << p1.quantity() << endl;
  return 0;
  }
```

程序的运行结果如图9-7所示。

```
Point created: x=640 y=0
Number of points: 1
Point created: x=0 y=0
Point created: x=30 y=0
Point created: x=50 y=70
Number of points: 4
Point destroyed: x=50 y=70
Point destroyed: x=30 y=0
Point destroyed: x=0 y=0
```

图9-7　程序9-6的输出结果

下面让我们再来看另一个版本的Point类，它增加了一个成员函数，用来比较两个Point参数的坐标，并返回两个点的坐标是否相等。

```
class Point {
     int x, y;                            // private coordinates
     static int count;                    // private count of objects
  public:
     Point (int a=0, int b=0)             // versatile constructor
       { x = a;  y = b;  count++; }
     static int quantity()                // it cannot be const
     { return count; }
     bool samePoints (const Point &p1, const Point &p2)
       { return p1.x == p2.x && p1.y == p2.y; }
  . . . } ;
```

上面的代码向维护人员传达了足够的设计信息吗？没有！一个明显的不足是，代码没有反应这个事实：函数samePoints()没有改变目标对象的状态，该函数应该被定义为const。但这只是其中一个问题，还有一个问题是，我们应该告诉维护人员这个函数只对其参数进行了操作。该函数可以成为全局函数，因为它并不需要目标Point对象的数据成员，它只是对参数的数据成员进行操作而已。我们将函数samePoints()定义为类Point的成员函数只是表明它逻辑上属于类Point，它处理的只是类Point的对象而不是类Rectangle、类Circle或者其他类的对象。因此，该函数应该被定义为静态成员函数。

为了说明这个问题，我们看看如何调用这个函数。有几种调用方法。下面是一个例子。

```
Point p1, p2(30), p3(50,70);
if (p1.samePoints(p2,p3)==true) cout << "Same points\n";
```

对象p1和进行比较的对象p2、p3有什么关系吗？没有。这样的代码很不美观。另一个用法是使用两次对象p2。

```
Point p1, p2(30), p3(50,70);
if (p2.samePoints(p2,p3)==true) cout << "Same points\n";
```

代码仍然不美观。对象应该只被使用一次。让我们把函数定义为静态函数。

```
class Point {
     int x, y;                              // private coordinates
     static int count;                      // private count of objects
  public:
     Point (int a=0, int b=0)               // versatile constructor
       { x = a;  y = b;  count++; }
     static int quantity()                  // it cannot be const
     { return count; }
     static bool samePoints (const Point &p1, const Point &p2)
       { return p1.x == p2.x && p1.y == p2.y; }
  . . . } ;
```

现在使用类作用域运算符调用这个函数。

```
Point p1, p2(30), p3(50,70);
if (Point::samePoints(p2,p3)==true) cout << "Same points\n";
```

这样的代码看起来就舒服多了。

那么应该在什么时候使用静态数据成员和静态函数呢？我们定义数据成员为静态变量，以表明此全局数据逻辑上属于该类（如count）。我们定义成员函数为静态函数，以表明此全局函数逻辑上属于该类，而且该函数只对静态数据、全局数据或者参数进行操作，而不对非静态数据成员进行操作。（例如quantity()、samePoints()）。

在学习使用C++时，静态数据和函数并不是首要的。但是，一定要理解底层的思想，用它们来把开发时的思路传达给客户代码程序员和维护人员。

9.7 小结

在这一章中，我们学习了如何使用C++的类这个程序开发工具。使用类消除了使用全局函数作为面向对象程序设计机制的缺点。

使用全局函数的第一个不足是，它不能准确地向维护人员表达设计人员在编码时候的意图，也不能告诉维护人员这些函数访问逻辑上属于一起的相同数据结构。

例如，程序使用了Point和Rectangle数据结构，设计人员把所有访问Point的函数放在一起，把所有访问Rectangle的函数放在一起。如果这些访问函数分开放，编译程序不会有什么问题，但维护人员会觉得糊涂。因此，程序员的修养是很重要的。

第二个不足是使用全局函数进行封装是自发的。程序员可以忽视访问函数而直接访问结构的域。语言规则不会阻止我们这样做。因此，这又依赖于程序员的修养了。

第三个不足是所有函数都是全局的，它们的名字是全局命名空间的一部分，可能和其他

函数名冲突。因此，为了避免冲突，程序员必须知道在工程中使用的所有函数名，这就增加了程序员开发系统的复杂性。

C++的类解决了以上3个问题。通过把数据和操作绑定在一起消除了第一个不足；通过控制对数据域的访问消除了第二个不足；通过使用类作用域消除了第3个不足。

总之，使用类可以大大提高软件的质量。

第10章 运算符函数：另一种好设计思想

前几章，我们讨论了C++类的语法和语义。C++并不是第一个支持类概念的程序设计语言，但它是第一个成功地运用于大型工业软件开发的语言。

开始，人们接受C++比较慢，因为业界怀疑它的效率和健壮性。担心效率是毫无根据的，绝大多数的C++程序占用的内存和C程序占用的内存差不多。绝大多数的C++程序的执行速度和C程序也差不多。当然，除了涉及到iostream库、虚函数、模板（这些将在后面几章讨论）的使用以外。但计算机硬件能力的巨大进步导致了计算机内存容量的极大增加和大多数计算机执行速度的飞跃。这就缓解了对C++内存需求和运行时性能的担忧。C++的使用经验清楚地证明了用类进行程序设计是一个很好的提高效率的途径。将要开发的任何新的编程语言都应该能支持类。

程序的健壮性问题则没有很好地从根本上解决，而是采用了另一个方法，用业界的经验教训指出程序员应该注意的危险和陷阱。奇怪的是，这并没有阻碍C++成为广泛使用的主流程序设计语言。语言的复杂性是导致程序健壮性降低的主要因素。在前一章，我们看到C++类背后的程序设计思想是很简单的，C++类能帮助程序员：

- 将对象数据和操作绑定在一起。
- 控制从类外部对类元素的访问。
- 为避免名字冲突引入额外的作用域。
- 将职责从客户推到服务器。

前一章还说明了C++的设计者Bjarne Stroustroup让C++类不仅实现了上述4点需求，还提供了其他的优点。构造函数和析构函数帮助类对象管理其资源，这些资源大多数是动态分配的内存。使用成员函数增加了类程序员（服务器设计人员）的负担，他们需要提供不同的构造函数在不同的上下文中支持客户。这也给客户端代码程序员提供了额外的负担，因为他们必须为对象初始化提供数据，然而我们把这种影响看做是不重要的。

使用复合对象也导致额外的复杂性。容器类的设计人员应该考虑其组成对象初始化的方便性问题，成员初始化列表应该提供这样做的新语法。复合类的概念也需要结合诸如常量、引用、指针和递归等其他组成成分。类属性的概念又扩展了这个思想，例如静态数据成员和静态函数，它们将类刻画成一个整体，而不是单个的类对象实例。

我们还提到C++的另外一个设计目标：以对待内部数据类型变量相同的方式对待类实例。在前一章，对象和变量初始化的统一语法体现了这个原则。在本章，我们也会讨论该设计原则的另一个体现：在C++运算符中应用这一原则，这样，运算符的相同语法也可以应用于类对象，其方式与应用在传统C++表达式中的内部数据类型变量上一样。

C++依然提供多种方法实现以上功能。我们会讨论实现运算符函数重载的不同技术。这些技术有助于我们更加有效地使用C++，也有助于我们理解C++程序的低层思想。

10.1 运算符重载

在C++中，程序员定义类型（特别是类类型）是内部数字类型概念的扩展。我们可以定义

程序员定义类型的变量，使用的语法和定义简单数据变量的语法一样。与使用基本类型相似，我们可以使用程序员定义类型的对象实例作为数组的元素，或者作为更加复杂的类型的数据成员。我们可以像内部数据类型那样，把程序员定义类型的对象当做函数的实际参数传递或者从函数返回它们。我们也可以使用与内部类型同样的语法设置指向程序员定义类型的值的指针和引用。也可以将指针定义为常量指针。还可以用和内部类型同样的语法，定义指针和引用为指向常量值的指针和引用。

这些相似并非偶然，C++的设计目标之一就是以对待内部数据类型变量相同的方式对待类实例。这个设计目标和面向对象的编程思想毫不相关，也和提高软件的开发效率、增强可维护性或者其他软件工程思想无关。这完全是从美学的角度来考虑的，而且符合C++的语法。计算机程序和其他任何富有创意的工作一样需要美观。虽然很少会有C++的书介绍这一类的问题，但我们写出来的程序除了应该可读、可移植、可维护以外，还应该看起来舒服。

当然，很多程序，特别是大型的程序，并不追求代码的优雅和谐。它们甚至连可读性、移植性和维护性也不一定很好，但C++语言是设计来帮助程序员实现这些目标的。

尽管如此，按相同的方式对待类和内部数据类型还有一个很大的差距。程序员定义的C++类型并不和基本的数据类型完全一样。最大的不同就是我们不能对程序员定义类型使用C++运算符，如加、减、比较相等或者不等一类的运算。我们可以编写自己的函数来实现这些运算符，但是表示方法可能看起来会有点怪。

下面让我们来考虑一个简单的例子：通过实部和虚部来描述的复数。如果对复数不熟悉，可以把它看成是平面直角坐标系中的某一个点，实部对应x坐标而虚部对应y坐标。将复数进行加减运算的结果是另外一个复数，它的实部是两个操作数的实部的加减结果，虚部是两个操作数的虚部的加减结果。复数的乘除运算较为复杂，但也是对实部和虚部进行一定运算的结果。

我们将用一个包含两个数据成员的类来描述复数，数据成员是real和imag。为简单起见，我们先把这两个数据成员定义为公共的（在下一个版本中将会是私有的）。

```
struct Complex {
   double real, imag; } ;          // public data members
```

程序10-1的代码定义了类Complex的对象实例，初始化它们，然后对这些对象执行一些算术操作。

注意　这并不是一个好的C++代码的例子。大多数关于C++的书不会给出这样糟糕的代码，因此我们就很难对比出代码好坏之间的差别。这有点像学习绘画时只去博物馆参观大师的作品，而不上艺术课。和绘画相似，C++设计总是努力找到最好的解决方案。本书选择了一些不好的解决方案作为示例，这样可以解释错误的地方在哪里，如何去改进它，最后给出一个比较好的解决方案，并说明为什么这个方案更好。

在程序10-1中，客户代码直接访问公共的对象成员，并执行复数计算：客户代码直接使用数据成员名real和imag，而没有使用访问函数。因此，客户代码既访问了数据域，又对数据域值进行了计算。这些计算的意义没有在函数调用中体现出来，因此维护人员必须分析计算的低层细节才能推断出来。低层操作的职责没有推到服务器函数，而必须由开发人员同时记住算法的几个层次：计算的高层目标和低层细节。它们不是可以单独考虑的领域，对类Complex设计的改变会影响客户代码。在这里使用关键字struct而不是class会更加合适

一些，因为所有的数据成员都是公共的。

程序10-1 对类Complex的对象执行的操作例子

```
#include <iostream>
using namespace std;

struct Complex {                                    // programmer-defined type
   double real, imag; } ;

int main()
{  Complex x, y, z1, z2;                            // objects of type Complex
   x.real = 20; x.imag = 40;                        // initialization
   y.real = 30; y.imag = 50;
   cout << " First value:  ";
   cout << "(" << x.real << ", " << x.imag << ")" << endl;
   cout << " Second value: ";
   cout << "(" << y.real << ", " << y.imag << ")" << endl;
   z1.real = x.real + y.real;                       // add real components into z1
   z1.imag = x.imag + y.imag;                       // add imaginary components
   cout << " Sum of two values:       ";
   cout << "(" << z1.real << ", " << z1.imag << ")" << endl;
   z2.real = x.real + y.real;                       // add real components into z2
   z2.imag = x.imag + y.imag;                       // add imaginary components
   z1.real = z1.real + x.real;                      // add to the real component of z1
   z1.imag = z1.imag + x.imag;                      // add to the imag component of z1
   cout << " Add first value to z1: ";
   cout << "(" << z1.real << ", " << z1.imag << ")" << endl;
   z2.real += 30.0;                                 // add to real component of z2
   cout << " Add 30 to sum:          ";
   cout << "(" << z2.real << ", " << z2.imag << ")" << endl;
   return 0;
  }
```

程序的输出结果如图10-1所示。

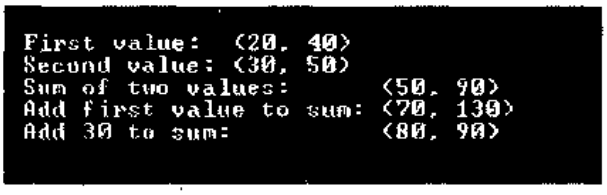


图10-1 程序10-1的输出结果

虽然这并不是一个面向对象程序设计的好例子，但它是我们进行运算符函数重载讨论的一个好的开端。而且，我们也借此机会再次列举出使用C++不当的缺点。这个缺点列表很重要，要不断地用这个列表评价我们的代码，这是学习如何正确使用C++和如何提高C++代码质量的最好方法。

为了对客户代码封装数据设计的细节，我们必须编写访问函数对Complex类型的对象进行操作，为客户代码服务。例如，我们想要让该类型的变量相加，就要编写一个函数，它接受两个输入Complex类型的参数，对这两个对象的成员执行必要的计算，再返回一个同样的Complex类型值的结果。这意味着这个命名（例如，addComplex()）函数的界面类似如下：

```
Complex addComplex(const Complex &a, const Complex &b);
```

就像我们在前面提到的一样，复数的加法就是把两个复数的实部和虚部分别相加。

```
Complex addComplex(const Complex &a, const Complex &b)
{ Complex c;                                    // local object
  c.real = a.real + b.real;                     // add real components
  c.imag = a.imag + b.imag;                     // add imaginary components
  return c; }
```

为了使用这个函数，客户代码要定义并初始化Complex类型的变量，并把它们作为参数传递给函数，然后将函数的返回值作为Complex类型的值使用。

```
Complex x, y, z1, z2;                   // objects of type Complex
x.real = 20; x.imag = 40;               // initialization
y.real = 30; y.imag = 50;
z1 = addComplex(x,y);                   // use in the function call
```

这样就好很多（而且简单）。大多数的程序员都习惯于这种函数类型的程序设计，不会觉得使用像addComplex()这样的函数名会让代码不够美观或者难读。但是，一个真正的C++程序员会觉得下面的用法更好。

```
Complex x, y, z1, z2;                   // objects of type Complex
x.real = 20; x.imag = 40;               // initialization
y.real = 30; y.imag = 50;
z2 = x + y;                             // use in expression
```

如果这样使用的话，编译程序会告诉我们加法运算还没有定义，尽管C++雄心勃勃要统一对待类型，但是这里还是不能真正统一对待。因为我们不能对程序员定义的数据类型使用基本运算，所以必须编写新的函数如addComplex()，并实现它们来执行必要的操作。程序员定义类型和基本类型之间的这种不统一对每个真正的C++程序员来说都是很痛苦的。

作为补救措施，C++提供了一种特别的函数。这种函数的命名有严格的规定，即在关键字operator后面加上要使用的运算符（如+等）。我们可以像设计和实现其他任何函数（例如addComplex()）那样设计和实现这个operator+()函数。C++允许我们使用与函数名中包含的运算符符号对应的运算符符号调用这个函数。例如，如果我们调用函数operator+()，就可以使用内部数据类型相同的语法进行函数调用。

```
z = x + y;              // under the hood, this is z = operator+(x,y);
```

这样的代码看起来就舒服多了，我们可以像普通的数据类型那样来操作对象，这看起来简直有点不可思议。

事实上，在C++中，同一作用域中的相同函数名可以代表不同的算法，只要它们的标识不同即可（参见第7章有关函数重载的讨论）。客户代码调用函数时，编译程序会用作用域中的所有函数声明与实际参数相比较，选择合适的函数，如果找到合适的函数就使用它以完成函数调用。

这对于任何C++函数名都是如此。对于算术运算符而言，运算符重载用在任何一种编程语言中而不只是C++中。运算符重载意味着给同一个符号以多个解释。如下面的加法运算符：

```
int a,b,c;
float d,e,f;
a = 20; b = 30;   d = 40.0; e = 50.0;
```

```
c = a + b;  f = d + e;        // different operations, same operator
```

在C++（和其他语言）中，运算符+用于整数或者是浮点数的加法运算。整数加法运算和浮点数加法运算在计算机内部是不同的。对于整数来说，将第二个操作数的每一位加到第一个操作数的每一位上，从低位到高位进位。

对于浮点数来说，其二进制表示包括尾数和指数。为了避免讨论二进制（或者十六进制）运算的复杂性，我们使用十进制系统中的例子。在尾数-指数表示方法中，如3000.0可以表示为3*10^3，300.0可以表示为3*10^2（在这里，我们使用运算符^来表示指数，虽然C++中没有指数运算符。）当两个浮点数相加时，较小的那个浮点数的尾数将会右移，以使两个浮点数的指数相同（如把3000.0和300.0相加时，300.0尾数将会右移3个小数位成为0.3*10^3），然后再把尾数相加（如3000.0和300.0相加，结果是3.3*10^3）。

无论浮点数加法的实现细节如何，但显然和整数加法运算的细节不同。从汇编语言的层次来看，这两种操作是由两条不同的指令来完成的。在高级语言中，我们并不强迫程序员学习整数加法和浮点数加法的不同表示符号。

希望大家在以上讨论中能够再次体会到信息隐藏的概念和把职责从客户代码推到服务器代码的思想。在上面的加法运算中，服务器是加法运算符，客户是使用有加法运算符的表达式的高层次代码。编写使用加法运算符的表达式的程序员无须了解加法的细节，而只用将精力集中在如何实现表达式的目标及其相关问题上。只有实现加法运算符的程序员才需要注意不同类型的加法细节，从而相应地实现每一个运算符。

C++允许使用相同的符号表示不同的运算符，并把这个功能扩展到程序员定义的数据类型上。如果我们遵守C++语法规则，就可以对任何程序员定义类型应用任何运算符（当然有个别例外情况）！

下面是为类型Complex的参数实现的operator+()函数。

```
Complex operator+(const Complex &a, const Complex &b)   // magic name
{ Complex c;                                            // local object
  c.real = a.real + b.real;                             // add real components
  c.imag = a.imag + b.imag;                             // add imag components
  return c; }
```

我们是如何编写这个函数的呢？我们复制了前面编写的addComplex()函数，保持函数体、返回类型和参数列表不变，删除函数名addComplex，把函数名重新定义为神奇的operator+。我们的任务就完成了。剩下的工作是C++的了：它会接受操作数为Complex类型的加法运算符，不会产生内容为类型Complex没有定义加法运算符的语法错误。该运算符现在已经被定义了。

```
Complex x, y, z;                    // objects of type Complex
x.real = 20; x.imag = 40;           // initialization
y.real = 30; y.imag = 50;
z = x + y;                          // use in expression
```

编译程序“会接受操作数为Complex类型的加法运算符”，这实际上是什么意思呢？编译程序会产生什么代码呢？编译程序会调用我们编写的重载函数operator+()，把表达式的左操作数作为函数的第一个实际参数，把右操作数作为函数的第二个实际参数。编译程序为上面的代码段产生的代码和为下面的客户代码产生的代码一样：

```
Complex x, y, z;                    // objects of type Complex
x.real = 20; x.imag = 40;           // initialization
y.real = 30; y.imag = 50;
z = operator+(x,y);                 // this is absolutely legitimate
```

如果函数名包含关键字operator和运算符符号，编译程序会认为它是函数调用语法或者运算符语法，并产生完全相同的代码。如果使用函数调用语法z=operator+(x,y)，编译程序和处理其他任何函数调用一样，匹配实际参数类型和形式参数类型。如果使用运算符语法z = x+y，编译程序会发现操作数是程序员定义类型，然后去搜索函数名中包含关键字operator和客户代码使用的运算符符号的函数。如果找到了这样的函数，编译程序检查函数的形式参数是否匹配客户表达式中的操作数个数和类型。

于是，职责就被推向服务器类中，客户代码从服务器设计的细节中解脱出来。客户端代码程序员可以对整数、浮点类型值、Complex类型对象或者其他任何程序员定义类型的加法使用相同的语法。

这是一种非常灵活而功能强大的机制。通过它，我们可以得到的好处比想像中还要多。开始时，我们的目标是以和处理基本类型变量相同的方法使用程序员定义类型，结果最后学习到更多的功能。现在，我们可以对自定义类型的对象进行一些特别的操作，这些操作我们从来没想过对基本的数值类型使用，因为C++语言没有限制我们使用自己的重载运算符函数。我们所受的限制只是在函数接口上——不能随意选择函数名和参数个数，而必须和重载的基本类型运算符保持一致。

程序10-2演示了运算符函数重载的使用。除了重载的加法运算符以外，程序也演示了operator+=()的使用，它把一个Complex对象加到另一个Complex对象上。程序也演示了另一个operator+=()的使用，它将一个浮点数加到Complex对象的实部。这两个运算符函数的名字虽然相同，但参数列表不同。这是函数名重载的合法使用（可参考第7章关于C++中函数名重载的详细讨论）。

程序10-2　运算符函数重载的例子

```
#include <iostream>
using namespace std;

struct Complex {                                    // programmer-defined type
   double real, imag; } ;

Complex operator+(const Complex &a, const Complex &b)       // magic name
{ Complex c;                                        // local object
  c.real = a.real + b.real;                         // add real components
  c.imag = a.imag + b.imag;                         // add imaginary components
  return c; }

void operator += (Complex &a, const Complex &b)     // another magic name
{ a.real = a.real + b.real;                         // add to the real component
  a.imag = a.imag + b.imag; }                       // add to the imag component

void operator += (Complex &a, double b)             // different interface
{ a.real += b; }                                    // add to real component only

void showComplex(const Complex &x)
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

```
int main()
{  Complex x, y, z1, z2;                          // objects of type Complex
   x.real = 20; x.imag = 40;                      // initialization
   y.real = 30; y.imag = 50;
cout << " First value:  ";  showComplex(x);
   cout << " Second value: ";  showComplex(y);
   z1 = operator+(x,y);                           // use in the function call
   cout << " Sum as function call:   "; showComplex(z1);
   z2 = x + y;                                    // use as the operator
   cout << " Sum as the operator:    "; showComplex(z1);
   z1 += x;                                       // same as operator+=(z1,x);
   cout << " Add first value to sum: "; showComplex(z1);
   z2 += 30.0;                                    // same as operator+=(z2,30.0)
   cout << " Add 30 to sum:          "; showComplex(z2);
   return 0;
  }
```

注意，在合适的地方使用了关键字const，如函数showComplex()、operator+()和第一个operator+=()中。注意，有些地方则没有使用const，如第一个和第二个operator+=()中。注意本例中一些面向对象程序设计的优点。客户代码不必依赖服务器的设计和数据域名（除了初始化外），低层计算的职责也推向服务器函数中。高层计算的意义表示为对服务器函数的函数调用。低层计算（根据复数算术处理复数的域）和高层计算（根据应用程序想要实现的目标处理复数）是不同的问题域。修改数据表示和应用程序语法也是单独的问题域：如果Complex类的设计改动了，只需修改重载的运算符函数无须修改客户代码；如果应用程序算法改动了，客户代码需要改变而重载运算符不需要改变。

但这个程序没有体现其他的面向对象优点：数据封装不是强制性的，没有什么表示数据和服务器函数属于一起，而且函数名是全局的。看来，我们要对程序做进一步的修改。

程序的运行结果如图10-2所示。

```
First value:  (20, 40)
Second value: (30, 50)
Sum as function call:   (50, 90)
Sum as the operator:    (50, 90)
Add first value to sum: (70, 130)
Add 30 to sum:          (80, 90)
```

图10-2 程序10-2的输出结果

从程序10-2可以看出，不论是否在关键字operator和运算符符号中使用了空格都是一样的：即operator+()和operator +()是等效的。如果运算符符号由两个字符组成，这两个字符应该连在一起而不能加上空格。

注意 使用重载运算符函数，必须在函数名中使用关键字operator和运算符符号。关键字operator和运算符符号一起组成函数名。如果为了提高可读性，可以在关键字和符号之间加上空格——用空格将函数名隔开来并不是语法错误。

在客户空间中使用重载运算符时，要记住它们是作为函数调用实现的。我们不能通过使用重载运算符来提高程序的执行速度，但可以增加程序的可读性。在所有使用重载运算符的地方，我们都可以用函数调用语法代替运算符语法。程序10-2的最后部分也可以写成如下形式。

```
operator+=(z1,x);                              // same as z1 += x;
cout << "Add first value to sum: " ; showComplex(z1);
operator+=(z2,30.0);                           // same as z2 += 30.0;
cout << "Add 30 to sum:          " ; showComplex(z2);
```

当然，我们编写重载运算符函数并不是为了在客户代码中用函数调用语法使用它们。如果想使用函数调用语法，还不如调用函数addComplex()而不是operator+()。我们费力地定义重载运算符函数，就是为了使用这个特别的属性，让C++编译程序将运算符语法作为函数调用来看待。在这里不断地提醒大家函数调用语法的目的是，一定不要忘记运算符语法会被编译成为函数调用形式，而不是外表上的算术表达式形式。

10.2　运算符重载的限制

在上一节中我们已经了解到，重载的运算符是让C++代码更加优美的强有力机制，它以相似的方式对待程序员定义类型对象和基本数值类型变量。但是，C++对重载运算符的使用设置了一些限制。有些限制对我们影响不大，但有些限制很重要。下面我们将详细讨论这些限制。

10.2.1　不可重载的运算符

有些对运算符重载的限制对程序员来说并不十分重要，至少在本阶段如此。我们不能重载的运算符有：运算符::（作用域运算符）、运算符.*（成员对象选择运算符）、运算符.（类对象选择运算符）、以及运算符?:（条件运算符或者算术if）。不知道是否有人可以解释为什么需要重载作用域运算符和条件运算符。同样，也没有必要重载成员对象选择符和类对象选择符。（实际上，成员对象选择运算符甚至还没有使用过。）

从实践的角度来看，这些限制比较重要的是，我们不能杜撰一些C++基本数值类型不支持的运算符。我们在重载运算符函数名中的关键字operator后添加的符号应该是真正的C++运算符。如果这个符号不是C++运算符就是语法错误。例如，C++没有提供指数运算符，而其他语言则用两个星号来表示指数运算，如FORTRAN中X**Y表示求X的Y次幂。有人可能想扩展C++的运算符集合，而重载双星号运算符。

```
Complex operator**(const Complex &a, const Complex &b);     // error
```

之所以出现错误，是因为C++不能从基本类型运算符中找出关于双星号运算符的定义。

我们也不能为内部数字类型重载运算符以赋予它们新的意义。例如，我们的应用程序可能想限制整数加法的和不能超过一个特定的数，例如60（取模运算）。我们就可能要重定义整数加法，让结果不超过60。

```
int operator + (int a, int b)                  // syntax error
{ return (a+b) % 60; }                         // addition modulo 60
```

这个主意看起来不错，但因为好几个原因而实际上行不通。主要原因是编译程序可能不能区分不同的加法意义，特别是应用程序重载了几个运算符的时候。如上面的例子为整数重载了加法运算符，但在函数体内也使用了加法运算符。我们希望函数体内的加法按普通意义计算，但是没有办法把这个信息告诉编译程序。编译程序怎么知道函数体内的加法不是一个调用新的重载运算符operator +()的递归函数呢？

客户代码中也有类似的问题：

```
int a,b,c;
float d,e,f;
a = 20; b = 30; d = 40.0; e = 50.0;

// built-in operator or overloaded operator?
c = a + b;    f = d + e;
```

在上面的代码中，我们就无从告知编译程序要使用基本类型的加法运算符还是使用重载的整数加法运算符。

这就是C++不允许为基本类型重载运算符的原因，我们只能为自定义类型使用重载运算符。事实上，C++将这个限制更概括为：要求重载运算符函数至少有一个参数是程序员定义类型（类）。我们刚才试图编写的整数加法运算符就违反了这个限制。

警告 我们不能通过重载不是基本类型C++运算符的运算符符号来扩展C++运算符，而只能重载已经存在的C++运算符。我们也不能以不同的方式为内部数据类型重载运算符，以改变内部数据类型的运算符的现有意义，我们只能为程序员定义类型（类）数据重载运算符。

注意，在所有的重载运算符函数中，我们设计的都是被重载的函数，而不是重定义现有的运算符。Complex对象的加法运算符不会消除整数和浮点数的加法运算符。重载的运算符被添加到C++编译程序理解的运算符列表中。我们再来看一下Complex的加法重载运算符：

```
Complex operator+(const Complex &a, const Complex &b)    // magic name
{ Complex c;                                              // local object
  c.real = a.real + b.real;                               // add real components
  c.imag = a.imag + b.imag;                               // add imaginary components
  return c; }
```

这个重载运算符函数体中的加法运算符是对浮点数进行标准加法运算的基本运算符。编译程序是如何知道这一点的呢？通过查看类Complex的数据域的类型，既然这些域都为double类型，函数体内的加法运算符就不是对正在定义的重载运算符的递归调用。客户代码也采取类似的分析思路。

```
Complex x, y, z;                          // objects of type Complex
x.real=20; x.imag=40;                     // initialization
y.real=30; y.imag=50;
z = x + y;                                // use in expression
double a, b, c;                           // variables of type double
a = 20; b = 30;                           // initialization
c = a + b;                                // use in expression
```

对于上面代码中的第一个加法运算符，编译程序确认操作数是类型Complex，于是调用重载的运算符函数。对于第二个加法运算符，编译程序辨别出两个操作数都是double类型，于是调用基本类型的加法运算符。

10.2.2 返回类型的限制

一般来说，重载运算符函数返回的值类型不是void、布尔值就是设计此运算符操作的类型值。返回一个类的类型值很常见，对于那些用于其他表达式中、计算出相同类型的新值的

运算符来说尤其普遍。例如前面例子中的operator+()函数返回Complex类型的值。这样就允许我们就对这个值使用赋值运算符。如果返回值是void类型，就不能用于赋值。而且，返回值允许我们编写与基本类型值的表达式相似的复杂表达式。

```
Complex a, b, c, d;                          // objects of type Complex
a.real=20; a.imag=40;                        // initialization
b.real=30; b.imag=50;
c.real = 0; c.imag = 20;
d = a + b + c;                               // use in expression
```

要理解这段代码的意义需要进行深入分析。C++的算术运算符是从左到右结合的。如果a、b和c是数字，表达式a+b+c的意义是(a+b)+c。重载运算符函数并不会改变运算符的结合性。当a、b和c是类型Complex的对象时，表达式的意义是相同的。

```
d = (a + b) + c;                             // use in expression
```

如果我们使用函数调用语法，这个表达式等效如下：

```
d = operator+((a + b),c);                    // use in expression
```

剩下的是再将表达式a+b用函数调用来表示。

```
d = operator+(operator+(a,b),c);             // use in expression
```

这段代码的意义是，将变量a和b作为实际参数调用函数operator+()，并将该函数调用的返回值作为对函数operator+()的再次调用的第一个参数。

我们在程序10-2中使用的两个重载运算符函数operator+()的返回类型是void，因此它们不能用于链式表达式中，因为链式表达式需要可以进一步运算的值。

```
Complex a, b, c, d;      // objects of type Complex
a.real=20; a.imag=40;    // initialization
b.real=30; b.imag=50;
c.real = 0; c.imag = 20;
d = a + b + c;           // use in expression
a += b;                  // OK: operator+=(a,b); returns void
d = c + (b += 30.0);     // not OK: operator+=(b,30.0); returns void
```

为了在链式表达式中使用这个运算符，我们必须重新设计为：

```
Complex operator += (Complex &a, double b)   // class return type
{ a.real += b;                                // add to real component
  return a; }
```

这就更好地模拟了内部数据类型的操作。本书并不特别喜欢C++中的内部数据类型的操作，因为它们容易导致程序员编写复杂的表达式而不是用简单的顺序的步骤描述算法。我们宁愿保留void的返回类型，将客户代码分解成不需要Complex返回值的子步骤，而不采用修改服务器代码（即重载运算符函数）来适应客户代码中的链式表达式的方法。

```
Complex a, b, c, d;       // objects of type Complex
a.real=20; a.imag=40;     // initialization
b.real=30; b.imag=50;
c.real = 0; c.imag = 20;
d = a + b + c;            // use in expression
a += b;                   // OK: operator+=(a,b); returns void
b += 30.0;                // OK: operator+=(b,30.0); return value not used
```

```
d = c + b;                    // OK: operator+=(c,b); returns Complex
```

但这只是风格问题。做上述比较的目的只是让大家明白在服务器和客户之间协作的方式有多种。

10.2.3 参数个数的限制

当我们设计一个重载运算符函数时，应该使用和该运算符（二元或一元）所必需的操作数相同个数的参数（通常是相同类类型）。

我们不能改变运算符的元，即使用运算符时需要指定的操作数的个数（一元运算参数个数为1，二元运算参数个数为2）。重载运算符的元数应该和原来的基本运算符的元数相同。我们不能定义一个作用于两个操作数的二元运算符，但将它用于创建对一个操作数进行处理的一元运算符。

下面是一个典型的违反这一规则的例子。我们试图重载小于运算符<，用它来实现程序10-2中showComplex()执行的输出Complex数据成员的操作

我们所需要做的只是删除showComplex函数名，替换成operator<。

```
void operator < (const Complex &x)            // not a good idea: syntax error
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

在客户代码中使用这个函数语法是很简单的：就像调用showComplex()函数一样。

```
Complex x, y, z1, z2;                 // objects of type Complex
x.real = 20; x.imag = 40;             // initialization
y.real = 30; y.imag = 50;
cout << "First value:  " ;
operator < (x);                       // same as showComplex();
cout << "Second value: " ;
operator < (y);                       // same as showComplex();
```

但是，小于运算符是二元运算符，使用这个函数的运算符语法需要有两个操作数，而上述代码漏掉了第二个操作数。

```
Complex x, y, z1, z2;                 // objects of type Complex
x.real = 20; x.imag = 40;             // initialization
y.real = 30; y.imag = 50;
cout << "First value:  " ;
< x;                                  // nonsense if x is numeric
cout << "Second value: " ;
< y;                                  // nonsense if y is numeric
```

运算符函数operator<()重载时需要两个参数，而我们只提供了一个需要打印的Complex类型的参数，所以上述代码是错误的。如果我们不知道第二个参数应该执行什么操作，就应该使用另一个只需要一个操作数的运算符。

C++有多个运算符既可以用作二元运算符也可以用作一元运算符：加、减、星号和@。将这些运算符重载为一元或者二元运算符都是可行的，因为两种情况对于内部数据类型都是合适的。例如，可以将运算符+重载为二元运算符。又因为这个运算符也可以作为一元加法符号，我们可以使用带一个参数的函数重载它。这样一来showComplex()的替换就是合法的。

```
void operator + (const Complex &x)            // same as showComplex()
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

现在，在客户端代码中使用运算符语法调用该函数就没有问题了。

```
Complex x, y, z1, z2;          // objects of type Complex
x.real = 20; x.imag = 40;      // initialization
y.real = 30; y.imag = 50;
cout << "First value:  " ;
+x;                            // operator+(x); or showComplex(x);
cout << "Second value: " ;
+y;                            // same as operator+(y); or showComplex(y);
```

10.2.4 运算符优先级的限制

重载运算符还有一个限制是不能改变运算符的优先级。

对于表达式x+y/2来说，无论我们把对象x和y声明为何种类型，也无论我们把运算符“+”和“/”定义为何种意义，“/”运算一定先于“+”运算。如果想改变这种顺序通常使用括号。

警告 当重载运算符函数时，我们不能改变这个运算符原有的操作数个数，或者改变运算的优先级或者结合性。我们所能做的就是为程序员定义数据类型定义运算符含义。这样就允许客户代码像标准C++基本数值类型那样对程序员定义类使用同样的表达式语法。

10.3 把重载运算符作为类成员

正如我们在前一章中所说的，任何和程序员定义数据类型相关的函数既可以实现为类成员函数又可以实现为全局的独立的非成员函数。对于算法和重载运算符函数来说也是这样。从实现类成员函数转向实现非成员函数，再转回实现成员函数，这是很重要的程序设计技巧。特别是对运算符函数很重要。

运算符函数可以定义为其参数所属类的成员函数。这时参数的个数比运算符的个数少一个（即二元函数有一个参数，一元函数没有参数）。缺少的参数成为使用该运算符时消息的目标对象。

10.3.1 用类成员取代全局函数

把重载运算符函数作为类成员函数的规则和作为非成员函数的规则一样。我们把成员函数的函数名改成关键字operator和正在被定义的运算符符号的串接。

例如，二元运算operator+()和二元运算operator+=()作为类Complex的成员函数实现时，应该只有一个参数，而不像程序10-2中作为全局函数实现的运算符那样使用两个参数。在函数接口中消失的那个参数的数据成员变成消息的目标对象的数据成员。

```
class Complex {                          // programmer-defined type
    double real, imag;                   // private data
public:
    Complex(double r, double i)          // general constructor
    { real =r;  imag = i; }
Complex operator+(const Complex &b)      // one parameter only
{ Complex c;                             // does it fly?
  c.real = real + b.real;                // add real components
  c.imag = imag + b.imag;                // add imag components
```

```
  return c; }
void operator += (const Complex &b)      // one parameter only
{ real = real + b.real;                  // add to the real component
  imag = imag + b.imag; }                // add to the imag component
                                         // THE REST OF CLASS Complex
};
```

希望大家能够很自然地从两个参数的非成员函数转到一个参数的成员函数。许多程序员喜欢使用两个参数的实现形式，因为它是对称的，我们可以将两个参数的对应域加起来。

```
Complex operator+(const Complex &a, const Complex &b)  // global name
{ Complex c;                                           // does it fly?

  // add components: symmetric notation
  c.real = a.real + b.real;
  c.imag = a.imag + b.imag;
  return c; }

void operator += (Complex &a, const Complex &b)        // global function
{ a.real = a.real + b.real;                            // add to the real component
  a.imag = a.imag + b.imag; }                          // add to the imag component
```

把上面第一个函数转换为成员函数时会遇到一个问题。我们使用了Complex类型的局部变量，但没有初始化它，因为我们并不关心它的数据成员是什么值——反正在函数结果返回给客户之前，函数会重写它的数据成员。程序10-2也使用了相同的设计，但是没有问题。因为我们没有为Complex类提供任何构造函数，所以系统会为该类提供一个什么也不做的缺省构造函数。但在上面的代码中，Complex类已经有了一个通用的构造函数，因此编译程序不会再提供缺省构造函数，而是认为运算符函数的第一行代码调用了一个不存在的函数。因此，我们应该随时考虑构造函数的问题。

对此，我们有两个补救方法。第一个方法是为局部对象提供我们并不需要的初始化值

```
Complex operator+(const Complex &b)    // one parameter only
{ Complex c(0,0);                      // a way to pacify the compiler
  c.real = real + b.real;              // add components: no symmetric notation
  c.imag = imag + b.imag;
  return c; }
```

更好的方法是去掉局部变量对象，而是创建一个未命名的Complex对象，用计算的结果直接对它进行初始化，最后从运算符函数中返回这个未命名对象的值：

```
Complex operator+(const Complex &b)                    // one parameter only
{ return Complex (real + b.real, imag + b.imag); }     // nice: fast and neat
```

注意 进行类设计时，我们不但要考虑到为客户代码提供支持，同时也要考虑到类本身各个成员函数的合理性。缺少必要的构造函数往往是类设计问题的根源。

在Complex对象的客户代码中，我们可以使用函数调用语法或者运算符语法。在下面的代码片段中，我们使用了两种调用形式。要注意，成员函数的函数调用语法和非成员函数的函数调用语法不同：有一个参数变成消息的目标对象。

```
Complex x(20,40),y(30,60),z1(0,0),z2(0,0);     // objects created
z1 = x.operator+(y);                           // use as the message to x
```

```
z2 = x + y;                                    // same as z2=x.operator+(y);
z1.operator+=(y);                              // use as the message to z1
z2 += x;                                       // same as z2.operator+=(x);
```

成员函数的运算符语法和非成员函数的运算符语法完全相同。编译程序负责以特殊的方式解释运算符语法的表达式，即将它视作一个函数，函数名包括关键字operator和对应的基本类型运算符的符号。

```
z2 = x + y;                          // same as z2=x.operator+(y);
z2 += x;                             // same as z2.operator+=(x);
```

我们也可以像对待全局函数那样采用函数调用语法来使用类成员函数。很少有程序员采用这种表示方法，我们在这里提及它只是为了说明表达式中运算符语法的真正含义。

```
z2=x.operator+(y);                   // same as z2 = x + y;
z2.operator+=(x);                    // same as z2 += y;
```

如果我们将同一个运算符既重载为全局函数又重载为成员函数，会出现什么后果呢？这不是好的做法。如果使用函数调用语法调用这些函数，编译程序可以分析出我们的意图。但是如果使用运算符语法，编译程序就会迷惑不解。两种函数都可以匹配，它们的优先级也一样，因此，编译程序会认为表达式有二义性而拒绝通过编译。

10.3.2 在链式操作中使用类成员

和非成员函数的实现相似，返回类型为void的成员函数可以排除在链式表达式中使用运算符语法。而返回类类型的对象的函数就可以参与链式表达式。

```
Complex a(20,40), b(30,50), c(0,20), d(0,0);      // defined and initialized
d = a + b + c;                                    // use in chain expression
```

基本类型运算符+是从左到右结合的，重载运算符+也是左结合的。因此上面的链式表达式的意义是d = (a+b)+c;。

我们实际上在处理一个发送给类类型Complex的实例的消息operator+()，二元运算符的语法模糊化了这个事实。对于成员函数的实现来说，a+b的意义是a.operator+(b)。因此，客户代码d = (a+b)+c;实际上等价于：

```
d = a.operator+(b) + c;     // message to return value of a.operator+(b);
```

第二个运算符+也表示传送给对象的消息，该对象被第一个函数调用返回。因此，链式表达式的意义是：

```
d=(a.operator+(b)).operator+(c);      //a message to the return value
```

希望大家明白如何将链式表达式解释成函数调用序列。

运算符的重定义只是发生在重载运算符被定义的类中。只有当消息（有一个Complex类型的参数）发送给一个Complex类型的对象时才使用新的定义。因此，成员函数定义中的c.real=real + b.real;语句使用的是+符号的标准定义，而不是对重载运算符+的递归调用，对double值应用的是基本类型的+运算符。编译程序能知道左操作数是double类型而不是对象，因此更不可能是operator+()消息的目标对象，所以就使用double值的基本运算符+。

程序10-3是新版本的Complex，它演示了作为成员函数实现的其他重载运算符。第二个

operator+=()函数没有Complex类型参数，而是使用了Complex目标对象。实现了showComplex()函数功能的一元函数operator+()现在根本没有参数了。这并没有和（作为非成员函数实现的）重载运算符函数应该至少有一个类参数对象的规则相抵触，因为这个参数对象现在成为消息的目标对象了。大多数程序员习惯重载运算符<<而不是运算符+来作为输出运算符。我们将在后面讨论如何做到。程序10-3的输出结果如图10-3所示。

程序10-3 作为类成员函数实现的重载运算符函数

```
#include <iostream>
using namespace std;

class Complex {                          // programmer-defined type
   double real, imag;                    // private data
public:                                  // public member functions

Complex(double r, double i)              // general constructor
   { real =r;  imag = i; }

Complex operator+(const Complex &b)      // one parameter only
{ return Complex (real + b.real, imag + b.imag); }   // fast and neat

void operator += (const Complex &b)      // does target object change?
{ real = real + b.real;                  // add to the real component of the target
  imag = imag + b.imag; }                // add to the imag component of the target

void operator += (double b)              // different parameter list
{ real += b; }                           // add to real component of the target

void operator + ()                       // it used to be showComplex(const Complex &x)
{ cout << "(" << real << ", " << imag << ")" << endl; } //

} ;                                      // end of class Complex

int main()
{  Complex x(20,40), y(30,50), z1(0,0), z2(0,0);     // objects created
   cout << " Value of x:  ";  +x;                    // same as x.operator+();
   cout << " Value of y:  ";  y.operator+();         // anything goes
   z1 = x.operator+(y);                              // use in the function call
   cout << " z1 = x + y:   ";
   +z1;                                  // display z1
   z2 = x + y;                           // same as z2=x.operator+(y);
   cout << " z2 = x + y:   ";
   +z2;                                  // display z2
   z1 += x;                              // same as z1.operator+=(x);
   cout << " Add x to z1:  ";  +z1;
   z2 += 30.0;                           // same as z2.operator+=(30.0);
   cout << " Add 30 to z2: ";  +z2;
   return 0;
   }
```

```
Value of x:  (20, 40)
Value of y:  (30, 50)
z1 = x + y:   (50, 90)
z2 = x + y:   (50, 90)
Add x to z1:  (70, 130)
Add 30 to z2: (80, 90)
```

图10-3 程序10-3的输出结果

10.3.3 使用const关键字

这里函数参数中使用的关键字const和程序10-2中的完全一样。但是程序10-2中的一些参数在程序10-3中消失了。下面是程序10-2中的全局服务器函数。

```
Complex operator+(const Complex &a, const Complex &b)      // magic name
{ Complex c;                                    // local object
  c.real = a.real + b.real;                     // add real components
  c.imag = a.imag + b.imag;                     // add imaginary components
  return c; }

void operator += (Complex &a, const Complex &b)            // another magic name
{ a.real = a.real + b.real;                     // add to the real component
  a.imag = a.imag + b.imag; }                   // add to the imaginary component

void operator += (Complex &a, double &b)        // different interface
{ a.real += b; }                                // add to real component

void showComplex(const Complex &x)              // it is operator+() in Listing 10.3
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

在程序10-2中，设计人员通过使用const关键字表传递函数operator+()的第一个参数的信息。与之类似，两个operator+=()函数的第一个参数也通过不使用const关键字表达了设计人员的设计思路。在程序10-3中，这些参数都不见了。我们怎么反映出这些对象的const关键字是否存在呢？这些对象并没有真正从应用程序中消失，而是从函数接口上消失了。如果在客户代码使用运算符语法就会看得特别清楚。

```
Complex x(20,40), y(30,50), z1(0,0), z2(0,0);      // defined, initialized
z2 = x + y;                                 // x and y do not change here
z1 += x;                                    // z1 changes as the result of operation
z2 += 30.0;                                 // z2 changes as the result of operation
```

无论这些运算符是实现为单独的非成员函数还是实现为成员函数，它们的右操作数都是不改变的，而左操作数作为运算的结果需要改变。查看这些运算符的函数调用语法并不能立刻清楚地得出这个结论。（记住：运算符语法只是一种可选择的形式，允许我们在遵守语言限制时使用。）

```
Complex x(20,40), y(30,50), z1(0,0), z2(0,0);      // defined, initialized
z2 = x.operator + (y);                      // x does not change during call
z1.operator += (x);                         // z1 changes as the result of operation
z2.operator += (30.0);                      // z2 changes as the result of operation
```

那么，我们应该如何表达这样的信息：在方法的执行过程中没有修改对象（消息的目标）的数据成员呢？这种情况下应该使用关键字const。但应该把const放在哪里呢？我们应该把它放在参数列表的闭括号和函数体的开花括号之间。在函数原型中，我们把它放在参数列表的闭括号和结尾分号之间。我们应该在任何可能的恰当的地方都使用const。

程序10-4和程序10-3相同，只是在适当的地方添加了const关键字，而且在类括号外实现了复数的成员函数。这就强迫我们在函数实现时使用类作用域运算符。作用域运算符在重载函数运算符中的使用方法和在任何其他成员函数中的使用方法一样。关键字const当然应

该在函数原型和函数实现中都出现。如果这两部分不一致将会出现语法错误（通常可能是条误导人的错误消息）。程序的输出结果与图10-3相同。

程序10-4 在类规格说明之外实现重载运算符函数

```
#include <iostream>
using namespace std;

class Complex {                                  // programmer-defined data type
   double real, imag;                            // private data
public:                                          // public member functions
  Complex(double r, double i);                   // general constructor
  Complex operator+(const Complex &b) const;     // no change to target
  void operator += (const Complex &b);           // target object changes
  void operator += (double b);                   // target object changes
  void operator + () const;                      // no change to target
} ;                                              // end of class Complex

Complex::Complex(double r, double i)             // general constructor
   { real =r;  imag = i; }

Complex Complex::operator+(const Complex &b) const
{ return Complex (real + b.real, imag + b.imag); }

void Complex::operator += (const Complex &b) // target changes
{ real = real + b.real;                          // add to real component of the target
  imag = imag + b.imag; }                        // add to imag component of the target

void Complex::operator += (double b)             // target object changes
{ real += b; }                                   // add to real component of the target

void Complex::operator + () const                // no change to target
{ cout << "(" << real << ", " << imag << ")" << endl; }

int main()
{ Complex x(20,40), y(30,50), z1(0,0), z2(0,0);     // defined, initialized
   cout << " Value of x:  ";  +x;                // same as x.operator+();
   cout << " Value of y:  ";  y.operator+();     // anything goes
   z1 = x.operator+(y);                          // use in the function call
   cout << " z1 = x + y:   ";  +z1;
   z2 = x + y;                                   // same as z2=x.operator+(y);
   cout << " z2 = x + y:   ";  +z2;
   z1 += x;  // same as z1.operator+=(x);
   cout << " Add x to z1:  ";  +z1;
   z2 += 30.0;                                   // same as z2.operator+=(30.0);
   cout << " Add 30 to z2: ";  +z2;
   return 0;
  }
```

提示 为没有修改其参数值的重载运算符函数的参数使用const关键字。将重载运算符实现为成员函数时，不要忘记为目标对象使用const关键字 如果目标对象没有改变，就一定要将函数标记为const。要确保没有const关键字就证明函数修改了目标对象。

10.4 案例分析：有理数

在这一部分中，我们将探讨运算符重载的另一个典型例子：一个封装了有理数（更确切地说是分数）并为有理数实现整数所支持的所有算术运算和比较操作的类。

有理数可以用两个部分表示：分子和分母。用分数进行运算可以避免用浮点数运算时的四舍五入误差，如1/4+3/2=14/8=7/4。

在类实现中，分子和分母都应该是私有数据成员。如果程序移植到16位机器上，这两个数据成员应该使用long类型；如果程序在32位机器上运行，这两个数据成员可能为int，也可能为long——因为32位机上的这两个数据类型表示同一数值范围。

```
class Rational {
  long nmr;
  long dnm;                    // private data
public:
  Rational()                   // default constructor: zero values
 { nmr = 0;  dnm = 0; }        // this is not a good idea
  Rational(long n, long d)     // general constructor: fraction as n/d
{ nmr = n;  dnm = d; }
                               // THE REST OF CLASS Rational
  } ;
```

一般构造函数使用客户代码指定的值初始化对象的域。缺省构造函数可以创建没有初始化的对象，而等待进一步的赋值，但是大多数程序员不喜欢让对象域未初始化，因此使用一些缺省值。如果这不影响程序性能是没有问题的。在本例中，我们将对象初始化为缺省的0值。

```
Rational  a(1,4), b(3,2), c, d;
c = a + b;        // 1/4+3/2 = (1*2+4*3)/(4*2)=14/8=7/4; c.nmr is 7, c.dnm is 4
```

缺省构造函数可以把分子分母的缺省值都设为0吗？如果一个对象只是用来作为表达式的左操作数而不是右操作数，像上面代码中的对象c那样，就没有什么问题。但如果客户端代码程序员假设未初始化对象总是被初始化为null，并将这个对象用于计算（例如累加求和），就可能带来问题。

```
Rational  a(1,4), b(3,2), c, d;
c = a + b;            // c.nmr is 7, c.dnm is 4
d += b;               // 0/0 + 3/2 = (0*2+3*0)/(0*2); d.nmr=0, d.dnm=0
```

因此我们应该为分母赋予一个非0初值，如1。

```
Rational::Rational()
{ nmr = 0;  dnm = 1; }      // zero value in the form 0/1
```

有了这样的缺省构造函数，Rational对象就可以用作左值和右值了。在当做左值使用时（如下面的例子中的对象c），构造函数的调用就徒劳无功了。

```
Rational  a(1,4), b(3,2), c, d;
c = a + b;            // c.nmr is 7, c.dnm is 4
d += b;               // 0/1 + 3/2 = (0*2+3*1)/(1*2); d.nmr=3, d.dnm=2
```

算术运算符可以作为重载运算符函数实现，它遵循分数运算的规则。下面的函数operator+()支持两个Rational对象的加法。函数代码的第一行注释表述了整个算法：即结果的分子是两个操作数（分数）交叉相乘的和，而结果的分母是两个操作数的分母之

乘积。

```
Rational Rational::operator + (const Rational &x) const
{ Rational temp;                    // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
  temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
  temp.dnm = dnm * x.dnm;           // for example, 1/4+ 3/2 = 14/8
  return temp; }
```

这个实现的问题是没有规范化结果，返有两个不好之处，首先是用户使用不方便；其次是分母在计算过程中只是增大，就很容易溢出。为了避免这种情况，Rational类应该支持化简算法，每次算术运算（包括对象构造）后都可以调用它。

```
class Rational {
  long nmr, dnm;                          // private data
public:
  Rational()                              // default constructor: zero value
 { nmr = 0;  dnm = 1; }
  Rational(long n, long d)                // general constructor: fraction as n/d
{ nmr = n;  dnm = d;
  normalize(); }
Rational operator + (const Rational &x) const   // important keyword
{ Rational temp;                          // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
  temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
  temp.dnm = dnm * x.dnm;                 // for example, 1/4 + 3/2 = 14/8
  temp.normalize();
return temp; }
void normalize()                          // find the greatest common divisor
{ if (nmr == 0) { dnm = 1;  return; }     // it is zero, no work to do
  int sign = 1;                           // make it -1 if the number is negative
  if (nmr < 0) { sign = -1; nmr = -nmr; }       // make both members positive
  if (dnm < 0) { sign = -sign;  dnm = -dnm; }
  long gcd = nmr, value = dnm;            // search for greatest common divisor
  while (value != gcd) {                  // stop when the GCD is found
    if (gcd > value)
      gcd = gcd - value;                  // subtract smaller number from the greater
    else  value = value - gcd; }
 nmr = sign * (nmr/gcd); dnm = dnm/gcd; }       // denominator is positive
// THE REST OF CLASS Rational
  } ;
```

喜欢算术的人可以跟踪一下化简算法的细节，而对那些对C++程序设计而不是算术更感兴趣的人来说，可以看看程序设计的问题

我们看到这里调用了两次normalize()函数。一次是在operator+()函数中，对局部变量temp应用这个操作。例如把1/4和3/2相加，结果是temp.nmr=14和temp.dnm=8。在进入while循环结构之前，gcd=14,value=8。在第一轮循环中（14>8），gcd=14-8=6,value=8。在第二轮循环中（6≤8），gcd=6,value=8-6=2。第三轮循环结束后，gcd=4,value=2。第四轮循环后，gcd=2,value=2，然后循环终止。实际上我们跟踪了整个算法（normalize()函数的运行过程如图10-4所示）。

第二次调用函数normalize()是在一般构造函数中。当客户代码如下所示实例化对象时，就需要进行简化。

```
Rational x(14,8);           // legitimate, but ugly
```

成员函数normalize()的目标对象是什么呢？在第9章我们化了很大的篇幅让大家理

解，成员函数和传统的全局函数看起来不同，被调用时使用的语法也不同。（即全局函数中的参数变成了消息的目标对象。）但在这里，没有任何目标对象，函数调用很像全局函数的调用形式。

	nmr	dnm	gcd	value	value != gcd　gcd > value
算法开始：	14	8			<（域的初始值）
循环前：			14	8	真：迭代　真：归约gcd
第1次通过后：			6	8	真：迭代　假：归约value
第2次通过后：			6	2	真：迭代　真：归约gcd
第3次通过后：			4	2	真：迭代　真：归约gcd
第4次通过后：			2	2	假：停止迭代
循环后：			2		<-（GCD最终值）
在算法终止前：	7	4			<-（域的最终值）

图10-4　成员函数normalize()的运行过程

当作为消息目标的对象没有显式指定时，消息的目标就是调用该函数的对象（除非这个函数是全局函数）。在本例中，消息的目标是Rational对象x，normalize()函数在其算法中使用的就是这个对象的x.nmr和x.dnm域。

如果全局函数的调用和成员函数的调用看起来语法相同——即没有消息的目标，一些C++程序员会觉得不习惯。于是在调用全局函数时，他们使用全局作用域运算符::，在调用相同类的成员函数时，他们使用对象指针this。

为什么有些程序员不喜欢用同样的表示形式调用成员函数和全局函数呢？从语法上来说它们是完全正确的。编译程序会搜索类中定义的成员函数列表，如果找到匹配的函数，就检查函数接口并产生函数调用；如果在类中找不到匹配的函数，编译程序在作用域内的所有全局函数中再次进行搜索。

而维护人员和编译程序不同。如果代码可以直接指示维护人员，告诉它一个类使用的函数是成员函数还是全局函数，那么就有助于提高程序代码质量（并降低代码复杂性）。

注意　应该尽可能地将我们设计类时的意图传达给代码的维护人员。如果一个函数调用没有消息目标，应该指出它是调用一个类的成员函数（使用指针this作为目标）还是调用全局的非成员函数（使用全局作用域运算符::）。

在下一个版本的Rational类中，我们使用这些技巧在一般构造函数中调用normalize()，在normalize()函数中调用返回long类型绝对值的全局函数labs()。我们还把normalize()函数从Rational类的公共部分转移到私有部分。

如果我们把normalize()定义为public成员函数，就等于是告诉客户端代码程序员，编写一个产生Rational对象的非规范化状态的算法也是可以的，而且该算法应该在客户代码中被使用。然而，我们添加normalize()函数的动机恰恰相反。它的目的是让客户端代码程序员从化简分数的职责中解脱出来，而把这个职责下推给服务器类Ratioanl。如果把这个函数定义为public，就会鼓励客户端代码程序员使用这个函数，从而造成对类设计的依赖性。这几乎和将数据成员设置为public一样糟糕。

正是因为这个原因，类的设计人员有个很重要的任务就是研究潜在的客户需求，提供必须而又不多余的服务。类提供的服务集合被称为类的公共界面。公共界面应该尽可能地精简，

同时不影响向客户代码提供足够的自解释性服务，也不会导致客户代码依赖服务器类的内部设计。

```
class Rational {
  long nmr, dnm;                          // private data
void normalize()                          // private member function
{ if (nmr == 0) { dnm = 1;  return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = ::labs(nmr); }// to illustrate it
  if (dnm < 0) { sign = -sign;  dnm = ::labs(dnm); }
  long gcd = nmr, value = dnm;      // search for greatest common divisor
  while (value != gcd) {            // stop when the GCD is found
    if (gcd > value)
      gcd = gcd - value;            // subtract smaller number from the greater
    else  value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }      // denominator is positive
public:
  Rational()                        // default constructor: zero values
  { nmr = 0;  dnm = 1; }
  Rational(long n, long d)          // general constructor: fraction in the n/d
  { nmr = n;  dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }
                                    // THE REST OF CLASS Rational
  } ;
```

对Rational类做的另一个重要改动与函数operator+()相关：我们去掉了对normalize()的调用，而是把运算的结果作为参数传递给Rational构造函数。前一个版本的Rational类中使用的运算符函数开销很大。这里我们再现这段代码。

```
Rational Rational::operator + (const Rational &x) const
{ Rational temp;                    // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
  temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
  temp.dnm = dnm * x.dnm;           // for example, 1/4 + 3/2 = 14/8
  temp.normalize();
  return temp; }
```

如果使用这样的重载运算符函数，下面代码段中的第二条语句要调用多少次函数呢？

```
Rational  a(1,4), b(3,2), c, d;
c = a + b;                          // c.nmr is 7, c.dnm is 4
```

如果回答“没有调用任何函数，只是两个分数相加而已”，就错了。这里没有加法运算；我们使用的是重载运算符，因此调用了函数。让我们以显式的函数调用的语法重写这段客户代码。

```
Rational  a(1,4), b(3,2), c, d;
c = a.operator + (b);               // c.nmr is 7, c.dnm is 4
```

现在，我们可以弄清楚至少有一个函数调用了。但再看看operator+()的函数体，可以看出在创建对象实例temp时调用了Rational的构造函数吗？还可以看出也调用了normalize()吗？这里已经有3个函数调用了。

然后，在函数返回程序员定义类型的值时，要创建一个新的类的未命名对象，并且调用拷贝构造函数从现有的对象（本例中是temp）的域初始化新对象的域。最后在表达式c =

a.operator+(b);中执行赋值运算符，这也等同于一个函数调用。于是又添加了两个函数调用，——在我们认为只有一个函数调用的地方实际上一共进行了5次函数调用。

但我们还没有分析完。函数执行到闭花括号时函数终止，所有的局部变量（在本例中指的是temp）都被撤销。对象实例被撤销时会做什么呢？应该是调用析构函数。而且，在客户空间中执行赋值以后，用来从函数返回值的未命名对象（使用拷贝构造函数初始化）也要被撤销，于是再次调用析构函数。到此已经有7个函数调用了。

大家可能希望运算符函数的新版本可以把函数调用的个数降到一个或者两个，而事实上不太可能。新版本消除了对变量temp的构造函数和析构函数的调用、对normalize()函数的调用和对返回值的拷贝构造函数的调用。但增加了对一般构造函数的调用以及在该函数内部对normalize()的调用。所以共调用了5个函数。这个改进好像没什么大变化，但我们要明白很多东西都是一点一滴积聚起来的。

要学会在编写C++代码时看到隐藏的函数调用。这里调用两个函数，那里又调用了两个函数，我们的程序就会劳而无功。因此，C++程序员不喜欢从函数返回对象，尽管这在C++中是合法的。

警告　要注意C++代码中的构造函数和析构函数调用。避免不必要的函数调用，避免从函数中返回对象。只有在为了支持客户代码中必要的语法形式时才这样做。

Rational类还应该为其客户提供什么其他服务呢？除了operator()+以外，还应该为其他三个算术运算operator-()、operator*()、operator/()实现重载运算符函数。和我们在本章前面讨论的Complex类相似，每个算术运算符函数应该返回该类类型的值。结果值可以被赋值给该类型的另一个值，或者用作链式符号中的消息目标。

数值算法通常需要在值之间进行比较，类Rational也不例外。条件测试通过时，重载比较运算符应该返回true（或1），否则返回false（或0）。

```
bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
{ return (nmr * other.dnm > dnm * other.nmr); }
```

也可以类似地重载其他条件运算符。注意，我们在上述代码中很小心地指出这些函数都不会改变其参数值，也不会改变目标对象的值。能够区别这两者的不同吗？

程序10-5显示了类Rational的实现，并含有用来演示类所支持的操作的测试代码。这是C++设计的一个好例子，重载运算符函数的使用反映了数值数据类型使用相同运算的方式。程序10-5的输出结果如图10-5所示。

程序10-5　类Rational和测试代码

```
#include <iostream>
using namespace std;

class Rational {
  long nmr, dnm;                          // private data
  void normalize();                       // private member function
public:
```

```
  Rational()                                    // default constructor: zero values
  { nmr = 0;  dnm = 1; }
  Rational(long n, long d)                      // general constructor: fraction as n/d
  { nmr = n;  dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;      // constant target
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
  void operator += (const Rational &x);             // target changes
  void operator -= (const Rational&);
  void operator *= (const Rational&);
  void operator /= (const Rational&);
bool operator == (const Rational &other) const;     // constant target
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  } ;                                               // end of class specification

Rational Rational::operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator - (const Rational &x) const
{ return Rational(nmr*x.dnm - x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator * (const Rational &x) const
{ return Rational(nmr * x.nmr, dnm * x.dnm); }

Rational Rational::operator / (const Rational &x) const
{ return Rational(nmr * x.dnm, dnm * x.nmr); }

void Rational::operator += (const Rational &x)
{ nmr = nmr * x.dnm + x.nmr * dnm;          // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                        // n1/d1+n2/d2 = (n1*d2+n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator -= (const Rational &x)
{ nmr = nmr * x.dnm - x.nmr * dnm;           // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                         // n1/d1+n2/d2 = (n1*d2-n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator *= (const Rational &x)
{ nmr = nmr * x.nmr;  dnm =  dnm * x.dnm;
  this->normalize(); }

void Rational::operator /= (const Rational &x)
{ nmr = nmr * x.dnm;  dnm =  dnm * x.nmr;
  this->normalize(); }

bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
{ return (nmr * other.dnm > dnm * other.nmr); }
```

```
void Rational::normalize()                    // private member function
{ if (nmr == 0) { dnm = 1;  return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = -nmr; }     // just for illustration
  if (dnm < 0) { sign = -sign;  dnm = -dnm; }
  long gcd = nmr, value = dnm;                // find greatest common divisor
  while (value != gcd) {                      // stop when the GCD is found
    if (gcd > value)
      gcd = gcd - value;                      // subtract smaller number from greater
    else  value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }    // denominator is positive

void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

int main()
{ Rational  a(1,4), b(3,2), c, d;
  c = a + b;                                  // c.nmr is 7, c.dnm is 4
  a.show(); cout << " +"; b.show(); cout << " =";
  c.show(); cout << endl;
  d = b - a;
  b.show(); cout << " -"; a.show(); cout << " =";
  d.show(); cout << endl;
  c = a * b;                                  // c.nmr is 3, c.dnm is 8
  a.show(); cout << " *"; b.show(); cout << " =";
  c.show(); cout << endl;
  d = b / a;
  b.show(); cout << " /"; a.show(); cout << " =";
  d.show(); cout << endl;
  c.show();
  c += b;
  cout << " +="; b.show(); cout << " ="; c.show(); cout << endl;
  d.show();
  d *= b;
  cout << " *="; b.show(); cout << " ="; d.show(); cout << endl;
  if (b < c)
  { b.show(); cout << " <"; c.show(); cout << endl; }
  return 0;
  }
```

```
1/4 + 3/2 = 7/4
3/2 - 1/4 = 5/4
1/4 * 3/2 = 3/8
3/2 / 1/4 = 6/1
3/8 += 3/2 = 15/8
6/1 *= 3/2 = 9/1
3/2 < 15/8
```

图10-5　程序10-5的输出结果

许多设计人员认为像Rational这样复杂的类应该为客户提供严格的访问其成员的方法，实现相应的get()和set()函数。

```
long Rational::getNumer () const          // notice const
{ return nmr ; }

long Rational::getDenom () const
{ return dnm ; }
```

```
void Rational::setNumer (long n)          // no const
{ nmr = n; }

void Rational::setDenom (long d)
{ dnm = d; }
```

但我们不赞成增加上面的几个成员函数，这样会增加类设计人员和客户端代码程序员的工作量。一般来说，应该避免允许客户访问实现的细节（无论是受约束的还是没有约束的）。如果客户需要访问，就将数据成员定义为public。如果有理数的结构不变，它会总是包含至少两个域。分母和分子都不会消失，域的名字也不会改变，因为其他名字并不会提供额外的益处。如果需要添加其他域，也不会使现有的直接访问分子和分母的代码无效。

注意 通常把数据成员置设成private，而把成员函数设置成public。如果局部函数只被类成员函数使用，要毫不犹豫地把这个局部函数定义为private。如果客户代码需要访问数据成员，而且类设计是稳定而不会变动的，就不会提供set()和get()访问函数，把数据成员设置为public就可以了。

以上的说法似乎与封装、信息隐藏、将职责推向服务器等原则相抵触。现在质疑private数据成员很可能会让人觉得可笑。

的确，使用public数据成员通常像在商业会议上打领带一样会与标准格格不入，但是并不总是这样。一定要根据实际情况判断什么才是合适的做法。

注意 如果一个类有设计良好和易于理解的数据成员，定义这些数据成员为public是可行的。例如几何类和代数类Point、Rectangle、Line、Complex、Rational等。

无论我们怎么设计这些几何类和代数类，它们的数据成员都不会消失，因此客户代码也不会改变。如果希望通过添加更多的成员函数提供更多的服务，也是很简单的。当然，不加区别地随便使用公共数据成员也会让类更加难以修改。

10.5 混合参数类型

类Rational和Complex都是很好的例子，演示了程序员定义类型是如何模拟C++内部数据类型的属性的，可以对这些类型的对象使用和普通数值变量相同的运算符集合。这正是C++的目标之一，即用同样的方法对待C++内部数据类型和程序员定义类型。

但这种模拟并不全面。有些运算符可以应用到数值类型变量上，而不能应用于Rational和Complex对象。这些运算符包括求余数运算、逻辑按位运算、逻辑移位等。当然，我们也可以像算术和比较运算符那样重载这些运算符，但是这些运算符（无论我们决定用这些运算符实现什么功能，编译程序都会通过）的意义对客户端代码程序员和维护人员来说并不直观。一个随意定义运算符意义的例子是前面的Complex::operator~()函数，我们用它来显示Complex对象的数据成员的值。从直观上来说，表达式+x应该对Complex类型的变量x执行什么操作并不清楚。

把基本类型对象和程序员定义类型对象等同看待的另一个问题是隐式类型转换。C++不遗余力地支持类型转换。下面的表达式对任何数值类型在语法和语义上都是正确的。

```
c += b;        // ok for b and c of any built-in numeric types
```

```
c += 4;        // ok for c of any built-in numeric type
```

无论变量b是什么数值类型，都会被隐式地转换为变量c的数值类型；无论变量c是什么数值类型，整数4都会被隐式地转换为c的类型。如果以上代码段中的变量b和c是Rational类型，第二行代码就会出现语法错误。若要它在语法上正确，客户代码中应该有下面函数中的任意一个。

```
void Rational::operator+=(int x);          // c+=4;  is  c.operator+=(4);
void operator+=(Rational &r, int x);       // c+=4;  is  operator+=(c,4);
```

但这两个函数都没有在程序10-5中实现，因而导致客户代码的语法错误。下面是一个可以避免此错误的成员函数的例子。

```
void Rational::operator += (int x)        // target object changes
{ nmr = nmr + x * dnm;                    // n1/d1 + n = (n1+n*d1)/d1
  this->normalize(); }
```

注意，如果我们同时提供上述的两个函数，即同时有这些函数接口的一个成员函数和一个全局函数，第二行代码仍然是错误的。这次是因为函数调用的二义性。两个函数都匹配（即都可以解释c+=4;），所以编译程序不知道应该调用哪个函数。

但是，如果变量b和c都是Complex对象，则c+=b;和c+=4;都是语法上正确的。为什么呢？因为程序10-4中实现了两个不同版本的operator+=()函数。

```
void Complex::operator += (const Complex &b);
void Complex::operator += (int b);
```

在下面的客户代码中，第一行代码调用上面的第一个函数，第二行代码调用上面的第二个函数。

```
c += b;        // c.Complex::operator+=(b); Complex argument
c += 4;        // c.Complex::operator+=(4); integer argument
```

这就解决了在表达式中使用混合类型操作数的问题。第二个重载运算符函数既可以作用于整数参数，也可以作用于字符、短整数、长整数、浮点数和双精度浮点数参数。根据参数转换规则，任何基本类型的值都可以转换为整数。所以没有必要为每个内部数据类型重载函数operator +=()。一个函数就足够了。

但不要以为万事大吉了，其他运算符，如-=、*=、/=等，情况又如何呢？这些运算符每个都需要另一个有一个数值类型参数的重载运算符函数。其他一些算术运算符函数，例如operator+()、operator-()、operator*()和operator/()又如何呢？我们来看下面关于类Rational的对象实例的代码：

```
c = a + b;     // c = a.operator+(b);
c = a + 5;     // ?? incompatible types ??
```

第二行代码又有语法错误，因为重载运算符需要一个Rational类型的对象作为实际参数，而不需要基本数值类型的值。同时，所有这些表达式并不是凭空想象出来的，在一些算法中数据值和复数、有理数的确互相混合。比较运算又如何呢？我们应该能够将Rational对象和整数相比较，而这又会产生其他的问题。

至今为止使用的解决方法都合法但是很繁琐。对每个以Rational（或者其他类）对象作为参数的运算符函数，我们都必须编写另一个以长整数值作为参数的运算符函数。（在16位

机器上整数可能还不足够。）

没有其他解决方法吗？有的，C++提供很好的工具，让我们可以只使用一个运算符（以类类型为参数）集合，然后强制这些运算符函数接受内部数值类型的实际参数。

这种工具是什么呢？它允许我们将数值类型值强制转换为类的值。下面来看一个简单的例子：

```
Rational c = 5;              // incompatible types ??
```

这行代码肯定是错误的。在第3章中，我们讨论过将一种内部数值类型的值强制转换为另一种内部数值类型的值的概念。当然，这种强制转换只能应用于内部类型之间，而不能应用于内部类型和程序员定义类型Rational之间。但如果存在内部数值类型和Rational类型之间的强制转换，其形式应该是怎样的呢？其语法和数值类型的语法一样：即将类型名放在括号中，这个类型名应该是值被转换的目标类型名。

```
Rational c = (Rational)5;    // this is how the cast should look like
```

在第3章中我们指出过，强制转换有两种语法形式，一种是从C语言中来的（即上面这行代码的形式），还有一种看起来像C++函数调用：

```
Rational c = Rational(5);    // this is how the cast could look like
```

上面的代码看起来是不是有点像构造函数？我们现在要怎么称呼这个产生类类型值的函数呢？为什么不称呼它为构造函数呢？因为这个函数看起来像构造函数，而且其行为也像构造函数：的确，这就是一个构造函数。

下一个问题，这是什么构造函数？这个问题就简单了，在第9章中，我们称只有一个非类类型参数的构造函数为转换构造函数。现在，我们已经理解为什么会使用这个名字（转换构造函数）。因为这个构造函数将参数类型的值转换为类类型的值。为了让上面那行代码在语法上正确，我们必须编写有一个参数的转换构造函数。

构造函数应该如何处理这单个参数呢？如果参数的值是5，Rational对象的值应该设置为5或者5/1。如果参数的值是7，对象值就应该设置为7/1。因此，参数的值应该用来初始化分子，而不论实际参数值是什么，分母都被设置为1。因此构造函数应如下所示：

```
Rational::Rational(long n)          // conversion constructor
{ nmr = n;  dnm = 1; }              // initialize to a whole number
```

当函数需要一个Rational参数而实际参数是数值类型时，都会调用这个构造函数。现在的类Rational代码如下：

```
class Rational {
  long nmr, dnm;                    // private data
  void normalize();                 // private member function
public:
  Rational()                        // default constructor: zero value 0/1
  { nmr = 0;  dnm = 1; }
  Rational(long n)                  // conversion constructor: whole value n/1
  { nmr = n;  dnm = 1; }
  Rational(long n, long d)          // general constructor: fraction as n/d
  { nmr = n;  dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }
```

```
                                        // THE REST OF CLASS Rational
};
```

如果能用一个有缺省参数的构造函数完成所有的工作，有些程序员就不喜欢编写多个构造函数。一个通用的构造函数可以用作一般构造函数、转换构造函数和缺省构造函数，如下所示。

```
Rational(long n=0, long d=1)        // general, conversion, default constructor
  { nmr = n;  dnm = d;
    this->normalize(); }
```

我们应该知道，当客户代码为对象初始化提供两个参数，或者一个参数，或者没有参数时，该构造函数都会被调用。在定义Rational对象时，并不是没有使用参数而是使用了缺省的参数值。

```
Rational a(1,4);      // Rational a = Rational(1,4); - two arguments
Rational b(2);        // Rational b = Rational(2,1); - one argument
Rational c;           // Rational c = Rational(0,1); - no arguments
```

注意，本例中提供的实际参数值是int类型，而构造函数需要long类型的参数。这是没有问题的，因为所有的内部数值类型之间都可以进行隐式转换，所以也可从int类型隐式转换到long类型。在函数调用中，编译程序不允许出现多于一次的基本类型转换，也不允许出现多于一次为类定义的转换（通过调用转换构造函数）。

在编译有Rational类型操作数的表达式时，编译程序首先把int类型实际参数转换为long类型，然后转换为Rational；转换后，编译程序调用合适的运算符。

```
c = a.operator+(Rational((long)5));       // real meaning of c = a + 5;
```

现在，即使没有运算符Rational::operator+(long)，上面的客户代码也可以通过编译。先创建临时的Rational对象，再调用转换构造函数，接着调用函数operator+()，然后调用Rational析构函数。

现在，我们可以编写以数值类型值为第二个操作数的客户代码，其第一个操作数还是Rational类型。

```
int main()
{ Rational  a(1,4), b(3,2), c, d;
  c = a + 5;                // c = a.operator+(Rational((long)5));
  d = b - 1;                // d = b.operator-(Rational((long)1));
  c = a * 7;                // c = a.operator*(Rational((long)7));
  d = b / 2;                // d = b.operator/(Rational((long)2));
  c += 3;                   // c.operator+=(Rational((long)3));
  d *= 2;                   // d.operator*=(Rational((long)2));
  if (b < 2)                // if (b.operator<(Rational((long)2))
    cout << "Everything works\n";
  return 0; }
```

程序10-6是类Rational的最新版本，它支持在二元表达式中使用混合类型。程序的输出结果如图10-6所示。

程序10-6　支持表达式中使用混合类型的类Rational

```
#include <iostream>
```

```
using namespace std;

class Rational {
  long nmr, dnm;                                    // private data
  void normalize();                                 // private member function
public:
  Rational(long n=0, long d=1)                      // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;      // const target
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
void operator += (const Rational &x);               // target changes
void operator -= (const Rational &x);
void operator *= (const Rational &x);
void operator /= (const Rational &x);

bool operator == (const Rational &other) const;     // const target
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  } ;                                               // end of class specification
Rational Rational::operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }
Rational Rational::operator - (const Rational &x) const
{ return Rational(nmr*x.dnm - x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator * (const Rational &x) const
{ return Rational(nmr * x.nmr, dnm * x.dnm); }

Rational Rational::operator / (const Rational &x) const
{ return Rational(nmr * x.dnm, dnm * x.nmr); }

void Rational::operator += (const Rational &x)
{ nmr = nmr * x.dnm + x.nmr * dnm;          // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                        // n1/d1+n2/d2 = (n1*d2+n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator -= (const Rational &x)
{ nmr = nmr * x.dnm - x.nmr * dnm;          // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                        // n1/d1+n2/d2 = (n1*d2-n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator *= (const Rational &x)
{ nmr = nmr * x.nmr;  dnm =  dnm * x.dnm;
  this->normalize(); }

void Rational::operator /= (const Rational &x)
{ nmr = nmr * x.dnm;  dnm =  dnm * x.nmr;
  this->normalize(); }

bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
```

```
{ return (nmr * other.dnm > dnm * other.nmr); }

void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

void Rational::normalize()                    // private member function
{ if (nmr == 0) { dnm = 1;  return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = -nmr; }
  if (dnm < 0) { sign = -sign;  dnm = -dnm; }
  long gcd = nmr, value = dnm;                // greatest common divisor
  while (value != gcd) {                      // stop when the GCD is found
    if (gcd > value)
      gcd = gcd - value;                  // subtract smaller number from greater
    else  value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }    // denominator is positive

int main()
{ cout  << endl << endl;
  Rational  a(1,4), b(3,2), c, d;
  c = a + 5;                                      // I'll discuss c = 5 + a; later
  a.show(); cout << " + " << 5 << " ="; c.show(); cout << endl;
  d = b - 1;
  b.show(); cout << " - " << 1 << " ="; d.show(); cout << endl;
  c = a * 7;
  a.show(); cout << " * " << 7 << " ="; c.show(); cout << endl;
  d = b / 2;
  b.show(); cout << " / " << 2 << " ="; d.show(); cout << endl;
  c.show();
  c += 3;
  cout << " += " << 3 << " ="; c.show(); cout << endl;
  d.show();
  d *= 2;
  cout << " *= " << 2 << " ="; d.show(); cout << endl;
  if (b < 2)
  { b.show(); cout << " < " << 2 << endl; }
  return 0;
  }
```

```
1/4 + 5 = 21/4
3/2 - 1 = 1/2
1/4 * 7 = 7/4
3/2 / 2 = 3/4
7/4 += 3 = 19/4
3/4 *= 2 = 3/2
3/2 < 2
```

图10-6　程序10-6的输出结果

要记住，虽然是隐式转换，然而到Rational类型的转换是通过调用转换构造函数来完成的。当函数终止时，转换中创建的临时对象也会随着析构函数的调用而撤销。（对本类而言，调用的是编译程序支持的缺省析构函数。）因此，与那些不依赖参数转换而是为每种类型的参数提供单独的重载运算符的版本相比，这个版本的类Rational执行得更慢一些。

通过转换构造函数实现隐式类型转换不但可用于重载运算符函数，也可以用于其他任何

函数，如成员函数、有对象参数的全局函数等。正如在第9章中提到的那样，转换构造函数削弱了C++的强类型系统。如果我们有意使用数值的值代替对象，这就没有什么。但是，如果无意中错用了，编译程序不会告诉我们犯了这个错误。

C++提供一个很好的防止出错，并强制客户代码的设计人员告诉维护人员有关信息的技术。这种技术就是在构造函数中使用关键字explic t。

```
explicit Rational(long n=0, long d=1)     // cannot be called implicitly
  { nmr = n;  dnm = d;
    this->normalize(); }
```

如果在构造函数中使用了explicit关键字，任何对该构造函数的隐式调用都会产生语法错误。

```
Rational  a(1,4), b(3,2), c, d;
c = a + 5;                          // syntax error: implicit call
c = a + Rational(5);                // ok: explicit call
d = b - 1;                          // syntax error: implicit call
d = b - (Rational)1;                // ok: explicit call
if (b < 1)                          // syntax error: implicit call
if (b < Rational(2))                // ok: explicit call
  cout << "Everything is fine\n";
```

这个主意不错，因为它允许类的设计人员可以很好地控制客户端代码程序员使用这个类对象的方式。

像Rational和Complex这样的程序员定义类，的确需要尽量模拟内部数值类型的行为在表达式中用数值类型值代替对象并不是错误，而是实现计算算法的合法技巧。在以上代码段中，我们将一些代码行标注为正确，而另外一些行标注为语法错误。如果可以选择的话，每个程序员都愿意使用被标注为语法错误的那种编码形式。在每个使用数值类型操作数的地方都要明确写出强制转换的语句，这个要求对客户端代码程序员来说太苛刻了，而且也导致代码不够美观。

从这个角度来看，为像Complex和Rational这样的类的构造函数使用关键字explicit显得有些多余了。

注意 不要对实现了重载运算符函数的数值类的构造函数使用关键字explicit。如果使用了explicit，在函数调用中试图使用基本类型参数代替该类的对象就会出现语法错误。

10.6 友元函数

让我们再回顾一下重载运算符函数，看看它是如何帮助我们以相似的方式对待程序员定义类型和数值类型，再看看有什么其他的做法。

正如我们开始所论述的那样，程序员十分希望以完全相同的方式对待基本类型的变量和程序员定义类型的变量。C++支持这种方式，但是需要程序员遵循一些规则。程序员必须放弃选择函数名的自由，因为函数名必须以关键字operator开头，紧接其后的是我们希望让自己的类对象使用的C++基本类型运算符的符号（或一些符号）。

还是有其他一些小的限制，允许我们可以做什么和不允许我们做什么。例如，我们只能

使用现有的C++运算符（不能杜撰出语言不认识的自己的运算符），我们不能改变运算符的相对优先级，也不能改变运算的结合性或者所需的操作数个数。但是这些限制影响不大。如果我们能够遵循以上规则，C++就可以辨识出将运算符作为函数调用使用的表达式。

C++同样允许我们以调用其他C++函数的方式调用重载运算符函数——即使用函数名（关键字operator加运算符符号），但很少程序员会这样使用。既然要使用函数调用的形式，为什么还要使用关键字operator呢？还不如编写一个普通函数，给它定义更具描述性的函数名，例如addComplex()或者addToComplex()等。在本章中，我们使用重载运算符函数的全名进行函数调用的惟一目的是：提醒大家不要忘记C++程序的低层意义。在表达式中使用的每一个重载运算符实际上都是函数调用，至少进行了一次函数调用。如果使用局部对象或者返回对象，使用运算符还需要调用这些对象的构造函数和析构函数。

只要遵循了重载运算符函数的函数头编写规则，我们可以在函数内进行任何操作。典型例子是，我们在程序10-4中为类Complex重载的operator+()函数。下面的客户代码的意义是什么呢？

```
Complex x(20,40), y(30,50);     // defined, initialized
+x;  +y;                        // same as x.operator+(); and y.operator+();
```

如果x和y是整数，第二行代码意义就很明显：即保持数据原有的正负符号不变。这个运算似乎并没有什么意义，但是代码的意义是清楚的。但如果是Complex对象就不能认为它意味着保持数值的符号不变。它可以表示任何其他意思，而在本例中，它表示：打印数据成员的内容。对许多类来说，数字可以使用的运算符并不能作用于类的对象。将对象作为数字看待就可能产生意义并不是很直观清楚的代码，就像使用加法符号表示输出操作一样。（我们会在稍后讨论一个更好的方法为对象的输入输出重载运算符。）这是个很严重的问题。

如果重载运算符函数作用于两个对象实例，则是简单易懂的。如果一个操作数是对象实例（消息的目标），而第二个操作数是数值类型，就会有问题——虽然使用的是运算符语法，但实际上是用不兼容类型的实际参数调用重载运算符函数。

在前一节，我们讨论了解决这个问题的两个方法。一个是将重载运算符函数的数目加倍：即为每一个有类类型参数的函数编写重载函数，该重载函数有相同的函数名并且其参数为数值类型。这个解决方法很好，但是会让类设计膨胀、进而使类更加难以理解。

另一个解决方法是为每个运算符（其参数为类类型）重载一个函数，但是必须保证该类有转换构造函数。转换构造函数将数值类型的值转换成类类型的值。当用两个类类型的操作数使用此运算符时，在调用重载运算符函数之前不会调用这个转换构造函数。如果第二个操作数（函数的参数）是数值类型，就会在调用重载运算符函数之前隐式地调用转换构造函数（或者如果在定义中使用了关键字explicit，就是显式调用）。这种解决方法将类的大小保持在可管理范围之内，但是每次使用数值类型值作为实际参数时都需要创建和撤销一个临时的类对象。这可能会影响程序性能。例如，下面代码段的第一行代码没有调用任何转换构造函数，但是第二行调用了。

```
Rational  a(1,4), b(3,2), c;
c = a + b;          // c = a.operator+(b); - match, no constructor call
c = a + 5;          // c = a.operator+(5); - conversion constructor is called
```

但是，关于在表达式中混用类型的讨论还没有结束。客户代码中的语句序列又如何呢？可以直接支持两个Rational对象相加。通过额外地调用转换构造函数可以将Rational对

象和数字相加。但是不支持将一个数字和一个Rational对象相加。

```
Rational  a(1,4), b(3,2), c;
c = a + b;          // c = a.operator+(b); - match, no constructor call
c = a + 5;          // c = a.operator+(5); - conversion constructor is called
c = 5 + a;          // syntax error: c = 5.operator+(a); is impossible
```

使用了重载运算符成员函数的表达式通常是发送给其左操作数的消息。因此左操作数应该是一个对象实例。上面的最后一行代码中，左操作数是整数，我们不能向整数发送消息。只有程序员定义类型的对象才能接受消息。但是，从将对象和数字同等看待的角度来说，最后一行代码应该和其他代码一样是合法的。因此，如果遵循平等对待内部数值类型和程序员定义类型的原则，系统应该支持最后一行代码。

如果我们想使用的函数接口和客户代码需要的函数接口不同，一个解决方法是创建外包装函数。外包装函数的函数名和我们希望使用的一样，它的界面符合客户代码的需求，其惟一目的是调用我们希望在客户代码中使用的函数。例如对类Rational的operator+()函数来说，外包装函数应该有相同的名字，但是可以接受一个数值类型的值作为它的第一参数

```
Rational Rational::operator + (int i, const Rational &x) const
{ Rational temp1(i);                        // conversion constructor
  Rational temp2 = temp1.operator+(x);      // overloaded operator
  return temp2; }
```

或者把代码优化一下：

```
Rational Rational::operator + (long i, const Rational &x) const
{ Rational temp(i);          // call to the conversion constructor
  return temp + x; }         // call to operator+(const Rational&);
```

然而，上面定义的成员函数是不能用的。因为它有三个参数：消息的目标、数值类型参数、对象参数。我们怎么可以把它们都放在一个函数调用中呢？

```
Rational  a(1,4), b(3,2), c;
c.operator+(5, b);                          // c + ???
```

重载运算符函数作为发送给它左操作数的消息。这意味着，最后一行代码的意思是对象c与某实体相加。但是我们的原意是将5和其他值相加，并将最后的结果放在对象c中。因此，这行代码的语法是错误的。我们再看看另一种形式。

```
Rational  a(1,4), b(3,2), c;
c = b.operator+(5, b);                      // c = b + ???
```

如果函数名中没有包含关键字operator，这是可行的。数值5会被转换为Rational对象，然后和对象b相加，其结果被复制到对象c中。这种情况下对象b似乎就不再是消息的目标了——该对象和运算没有关系。但是上面的代码中，函数名包含了关键字operator，语法就不再正确了。应该只有两个操作数，而不是三个。

实际上，抛弃目标对象并调用只有两个参数的函数会更好一些。

```
Rational  a(1,4), b(3,2), c;
  c = operator+(5, b);                      // c = 5 + b; ???
```

还记得程序10-2中为类Complex编写的第一个重载运算符函数吗？它不是类成员函数，而是全局函数。为了让以上代码可行，我们需要做的就是定义一个全局的外包装函数。

```
Rational operator + (long i, const Rational &x)    // not a class member function
{ Rational temp(i);          // call to the conversion constructor
  return temp + x; }         // call to Rational::operator+(const Rational&);
```

实际上，我们不仅去掉了作用域运算符，还去掉了用来表示函数体不会修改目标对象的域的const修饰符。这里没有目标对象，因此不需要检查其域是否被修改。

这是一个好的解决方法，但还有些局限性。以其他方式编写表达式时使用这个函数也是可以的，不管第一个操作数是否是数值类型。编写这个函数的一个好方法是去掉局部Rational对象，用第一个参数而不是在函数体内调用转换构造函数。

当我们用成员函数重定义二元运算符时，左操作数是隐式的，其形式为指针。

```
Rational operator + (const Rational &x, const Rational &y)
{ return x.operator+(y); }     // call to Rational::operator+(const Rational&);
```

注意，在这个函数体内使用表达式的语法是不恰当的。它会被解释成对我们正在定义的全局函数operator+()的递归调用。

```
Rational operator + (const Rational &x, const Rational &y)
{ return x + y; }              // recursive call to operator+(): infinite loop
```

有时候需要使用函数调用语法而不是运算符语法显式地调用类的成员函数，这就是一个例子。它允许有两个参数的全局函数operator+()调用（有一个参数的）类成员函数。

我们仔细研究了为这个全局函数设计界面的步骤，因为仔细跟踪这些步骤很重要。可以用成员函数或者全局函数实现相同的算法，许多C++程序员对此并不习惯。在第9章我们总结了转变的规则：全局变量有一个额外的类参数；成员函数没有这个参数，而是使用消息的目标对象作为参数。一定要熟悉这种转变。

现在，必须承认这个设计还是有问题。我们在开始时没有和其他论题同时讨论，是因为不想分散注意力，现在应该讨论这个问题了。在客户代码中使用运算符语法时，编译程序有两种解释表达式的方法：或者用一个参数调用类成员函数，或者用两个参数调用全局函数。如果表达式有两个对象实例或者一个对象实例、一个基本类型操作数（调用合适的转换构造函数），每个函数都可以提供对表达式的合法解释。当然，如果两个操作数都是内部数值类型，就会出现二义性——编译程序将表达式解释成基本类型运算符，而不是对重载运算符函数的调用。

```
Rational  a(1,4), b(3,2), c;
c = a + b;       // ambiguity: c = a.operator+(b); or c = operator+(a,b); ??
c = a + 5;       // c=a.operator+(Rational(5)); or c=operator+(a,Rational(5));
c = 5 + a;       // no ambiguity: c=operator+(Rational(5),a); no 5.operator+(a);
c = 5 + 5;       // no ambiguity: the built-in binary addition operator
```

很遗憾，这与第9章所描述的编译程序解析名字的算法相冲突。对非运算符函数来说，编译程序首先查看类成员函数，只有在类作用域中找不到匹配的名字时，才继续在文件所知的全局函数中查找。而对运算符函数不会遵循这个规则。

为了消除两个操作数都是对象的表达式的二义性，我们可以去掉成员运算符函数，直接使用全局函数实现这个运算符的算法。这样编译程序就只有一种方法解释表达式了。

```
Rational operator + (const Rational &x, const Rational &y)    // no Rational::
{ return Rational(y.nmr*x.dnm+x.nmr*y.dnm,y.dnm*x.dnm); }     // private data??
```

这个方法看起来不错，但过于直接了——它直接访问了参数的域，而函数本身是在类

Rational的作用域之外的，应该没有权力这样做。这就意味着这样的函数不能通过编译。

C++为我们提供了一种有趣的解决方法：使用友元函数（friend function）。友元函数是一个非成员函数，但却有和类成员函数相同的访问类成员的权力。注意，我们说的是“访问类成员的权力”，而不是“访问类数据的权力”，因为友元函数访问private（或者protected）成员函数和访问private（或者protected）数据成员一样容易。

友元函数可以是全局函数，也可以是另一个类的成员函数。实际上，在某些情况下，我们希望另一个类的所有成员函数都可以访问一个类的成员。这时，可以把另一个类定义为这个类的友元。（在第12章中会更加详细地讨论这种情况。）但是，大多数的友元函数是全局函数：如果想要把另一个函数的单个成员函数定义为这个类的友元，就要再考虑一下，因为这样可能会让事情变得更加复杂。

为了把一个函数定义为类的友元，必须把这个函数的函数原型插入到类的规格说明中（像普通的成员函数那样），然后在函数原型前面加上关键字friend。这个关键字才是问题的关键，否则从任何角度来看这个函数都像是这个类的成员函数。

```
class Rational {
  long nmr, dnm;                                  // private data
  void normalize();                               // private member function
public:
  Rational(long n=0, long d=1)                    // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
  friend Rational operator + (const Rational &x, const Rational &y);
                                                  // THE REST OF CLASS Rational:
                                                  // no need for operator+() functions
  } ;
```

友元函数和成员函数是不同的。调用友元函数时，不需要像调用类的成员函数那样指定目标对象。但是这个友元函数可以访问Rational类的成员，就像它是Rational类的成员函数一样。因此，下面这个版本的函数是完全合法的。

```
Rational operator + (const Rational &x, const Rational &y)    // no Rational::
{ return Rational(y.nmr*x.dnm+x.nmr*y.dnm,y.dnm*x.dnm); }     // yes, private data
```

用友元函数取代类成员函数就可以消除客户代码的二义性。

```
Rational  a(1,4), b(3,2), c;
c = a + b;   // no ambiguity: c = operator+(a,b);
c = a + 5;   // no ambiguity: c = operator+(a,Rational(5));
c = 5 + a;   // no ambiguity: c=operator+(Rational(5),a);
c = 5 + 5;   // no ambiguity: the built-in binary addition operator
```

有Rational对象的所有三种表达式形式都得到了支持。如果还担心调用Rational转换构造函数的问题，我们可以重载二次运算符函数，并将这三个函数定义为类Rational的友元函数，这样就可以避免调用转换构造函数。

```
class Rational {
  long nmr, dnm;                          // private data
  void normalize();                       // private member function
public:
  Rational(long n=0, long d=1)            // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
```

```
    friend Rational operator + (const Rational &x, const Rational &y);
    friend Rational operator + (const Rational &x, long y);
    friend Rational operator + (long x, const Rational &y);
                                    // THE REST OF CLASS Rational
    } ;
```

正如我们在前面提到的那样，可以对成员函数使用相似的多重载技巧。友元函数对于成员函数来说有一个优势，因为成员函数只支持Rational对象作为左操作数的形式，而不支持左操作数是数值类型变量的形式。因为对重载运算符成员函数的调用被解释成发送给左操作数的消息，支持左操作数是数值类型变量的形式就要求编译程序像下面这样解释表达式：

```
c = 5.operator+(a);      // an integer cannot respond to Rational messages
```

对于混合数值类型和对象类型操作数的表达式而言，友元函数具有更大的灵活性，因为它不要求左操作数为对象。

可以对Rational运算符进行类似的处理。参数为对象类型的类成员函数只支持以对象实例为操作数的表达式。如果想要支持将数值类型值作为右操作数的表达式，我们应该添加一个转换构造函数或者参数为数值类型的另一个重载运算符函数。但是，这样也仍然不能支持左操作数是数值类型值、右操作数是对象的表达式。

```
Rational  a(1,4), b(3,2);
if (a < b) cout << "a < b\n";      // a.operator<(b);
if (a < 5) cout << "a < 5\n";      // a.operator<(5);
if (1 < b) cout << "1 < b\n";      // 1.operator<(b); is nonsense
if (1 < 5) cout << "1 < 5\n";      // built-in inequality operator
```

在程序中添加全局的重载运算符函数，就可以支持上面的第四行代码。

```
bool operator < (const Rational &x, const Rational &y)
{ return x.operator<(y); }
```

与算术运算符类似，如果同时使用全局和成员运算符函数，第二行和第三行代码会有二义性。

```
Rational  a(1,4), b(3,2);
if (a < b) cout << "a < b\n";     // a.operator<(b); or operator<(a,b);
if (a < 5) cout << "a < 5\n";     // a.operator<(5); or operator<(a,5);
if (1 < b) cout << "1 < b\n";     // no ambiguity: operator<(1,b);
if (1 < 5) cout << "1 < 5\n";     // built-in inequality operator
```

为了支持所有形式的比较表达式，我们可以把每个成员运算符函数替换成可以直接访问其参数的数据成员的全局运算符函数。为了让这种数据访问合法，我们应该将这个全局运算符函数定义为类的友元。

```
class Rational {
  long nmr, dnm;                              // private data
  void normalize();                           // private member function
public:
  Rational(long n=0, long d=1)                // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
  friend Rational operator + (const Rational &x, const Rational &y);
  friend Rational operator - (const Rational &x, const Rational &y);
  friend Rational operator * (const Rational &x, const Rational &y);
```

```
  friend Rational operator / (const Rational &x, const Rational &y);
  friend bool operator < (const Rational &x, const Rational &y);
  friend bool operator > (const Rational &x, const Rational &y);
  friend bool operator == (const Rational &x, const Rational &y);
                                          // THE REST OF CLASS Rational
} ;
```

这样的设计就消除了二义性，可以支持各种形式的比较表达式：两个操作数都是对象，只有右操作数是对象、以及只有左操作数是对象（这是最难的情况）。

```
Rational  a(1,4), b(3,2);
if (a < b) cout << "a < b\n";      // operator<(a,b);
if (a < 5) cout << "a < 5\n";      // operator<(a,Rational(5));
if (1 < b) cout << "1 < b\n";      // operator<(Rational(1),b);
if (1 < 5) cout << "1 < 5\n";      // built-in inequality operator
```

我们可以看到，友元运算符函数可以完成成员运算符函数完成的相同工作，甚至还可以完成更多的其他工作。但有些运算符不能作为友元重载，如赋值运算符（operator=()）、下标运算符（operator[] ()）、箭头选择运算符（operator->()）以及函数调用或者括号运算符（operator()()）。这样的限制是为了确保参与运算的第一个操作数是左值（即消息的目标）。在本节前面的所有例子中，第一个操作数和第二个操作数都是右值。

接着，我们来看看算术赋值运算符。这个运算符的情况有些不同，因为它没有返回值（即返回值类型是void）、而只是修改了目标对象的状态。因为它没有返回Rational类的新值、就不会调用简化对象状态的Rational构造函数。因此，这个算术运算符必须在返回前调用Rational::normalize()函数。

```
void Rational::operator += (const Rational &x)          // no const
{ nmr = nmr * x.dnm + x.nmr * dnm; dnm = dnm * x.dnm;
  this->normalize(); }                                  // no constructor call
```

这个运算符函数支持的左右操作数都是对象的表达式（如c+=b;）、有了转换构造函数，该运算符函数也支持左操作数是对象而右操作数是数值类型值的表达式（如c+=5;）。

```
Rational  a(1,4), b(3,2), c;
c = a + b;          // c = operator+(a,b);
c += b;             // c.operator+=(b);
c += 5;             // c.operator+=(Rational(5));
5 += c;             // 5.operator+=(c); is nonsense, is it not?
```

初看起来，用全局重载运算符函数取代重载运算符成员函数似乎并没有用。

```
class Rational {
  long nmr, dnm;                    // private data
  void normalize();                 // private member function
public:
  Rational(long n=0, long d=1)      // general, conversion, default constructor
  { nmr = n;  dnm = d;
    this->normalize(); }
  friend void operator += (Rational &x, const Rational &y);    // no const!
                                    // THE REST OF CLASS Rational
} ;
```

这个运算符函数修改了第一个参数的值，因而参数中没有使用const修饰符。

```
void operator += (Rational &x, const Rational &y)       // no const!
```

```
{ x.nmr = x.nmr*y.dnm + y.nmr*x.dnm;  x.dnm *= y.dnm;
  x.normalize(); }                    // here, normalize() has the message target
```

我们在前面说过，友元函数可以访问所有的类成员，而不仅仅是类的数据成员。一致性对待数据成员在这里体现为：运算符函数可以访问其参数的私有数据成员和私有成员函数normalize()。

该函数支持的表达式形式和成员运算符函数支持的表达式形式一样。

```
Rational  a(1,4), b(3,2), c;  long x = 5;
c = a + b;            // c = operator+(a,b);
c += b;               // operator+=(c,b);
c += 5;               // operator+=(c,Raticnal(5));
5 += c;               // a constant cannot be used as an lvalue
x += c;               // operator+=(Rational(x),c); what is this?
```

向一个具体的数值值（常量值）添加任何值都是语法错误，这是勿庸置疑的。向一个数值类型变量添加值的情况则比较复杂。这个变量可能被修改，特别是作为参数传递给其引用类型参数没有使用const修饰符的函数时更是如此。

因为实际参数不是Rational对象，就需要进行类型转换。编译程序创建一个临时对象，并调用Rational的转换构造函数对它初始化，x的值就传递给该构造函数作为参数。接着，在运算符函数中修改这个临时对象（作为参数x）是没有用的，因为当函数终止时临时对象被撤销，这种改变不会还回给变量x。因此，编译程序声明x+=c;是语法错误。

即使在这种情况下，我们也不能同等地对待数值类型和程序员定义类型，本例证明了为达到这样的目标还有很长的路要走。事实上，许多程序员都喜欢使用全局友元运算符函数而不是成员函数，因为全局运算符函数更加容易编写——它对称地对待其运算符。

程序10-7是Rational类的实现，它将重载运算符函数实现为友元函数而不是成员函数。程序的输出结果如图10-7所示。

程序10-7　使用友元函数支持混合类型表达式的Rational类

```
#include <iostream.h>

class Rational {
  long nmr, dnm;                          // private data
  void normalize();                       // private member function
public:
Rational(long n=0, long d=1)              // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
friend Rational operator + (const Rational &x, const Rational &y);
friend Rational operator - (const Rational &x, const Rational &y);
friend Rational operator * (const Rational &x, const Rational &y);
friend Rational operator / (const Rational &x, const Rational &y);
friend void operator += (Rational &x, const Rational &y);
friend void operator -= (Rational &x, const Rational &y);
friend void operator *= (Rational &x, const Rational &y);
friend void operator /= (Rational &x, const Rational &y);
friend bool operator == (const Rational &x, const Rational &y);
friend bool operator < (const Rational &x, const Rational &y);
friend bool operator > (const Rational &x, const Rational &y);
void show() const;
  } ;                                     // end of class specification
```

```
void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

void Rational::normalize()              // private member function
{ if (nmr == 0) { dnm = 1;  return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = -nmr; }
  if (dnm < 0) { sign = -sign;  dnm = -dnm; }
  long gcd = nmr, value = dnm;          // search for greatest common divisor
  while (value != gcd) {                // stop when the GCD is found
    if (gcd > value)
      gcd = gcd - value;                // subtract smaller number from the greater
    else  value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }       // denominator is always positive

Rational operator + (const Rational &x, const Rational &y)
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }

Rational operator - (const Rational &x, const Rational &y)
{ return Rational(x.nmr*y.dnm - y.nmr*x.dnm, x.dnm*y.dnm); }

Rational operator * (const Rational &x, const Rational &y)
{ return Rational(x.nmr * y.nmr, x.dnm * y.dnm); }

Rational operator / (const Rational &x, const Rational &y)
{ return Rational(x.nmr * y.dnm, x.dnm * y.nmr); }

void operator += (Rational &x, const Rational &y)
{ x.nmr = x.nmr * y.dnm + y.nmr * x.dnm;  x.dnm *= y.dnm;
  x.normalize(); }

void operator -= (Rational &x, const Rational &y)
{ x.nmr = x.nmr*y.dnm + y.nmr*x.dnm; x.dnm *= y.dnm;
  x.normalize(); }

void operator *= (Rational &x, const Rational &y)
{ x.nmr *= y.nmr;  x.dnm *= y.dnm;
  x.normalize(); }

void operator /= (Rational &x, const Rational &y)
{ x.nmr = x.nmr * y.dnm;  x.dnm =  x.dnm * y.nmr;
  x.normalize(); }

bool operator == (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm == x.dnm * y.nmr); }

bool operator < (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm < x.dnm * y.nmr); }

bool operator > (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm > x.dnm * y.nmr); }

int main()
{ Rational  a(1,4), b(3,2), c, d;
  c = 5 + a;
  cout << " " << 5 << " +"; a.show(); cout << " =";
  c.show(); cout << endl;
  d = 1 - b;                                    // operator-(Rational(1),b);
```

```
cout << " 1 -"; b.show(); cout << " ="; d.show(); cout << endl;
c = 7 * a;                                          // operator*(Rational(7),a);
cout << " 7 *"; a.show(); cout << " ="; c.show(); cout << endl;
d = 2 / b;                                          // operator/(Rational(2),b);
cout << " 2 /"; b.show(); cout << " ="; d.show(); cout << endl;
c.show();
c += 3;                                             // operator+=(c,Rational(3));
cout << " += " << 3 << " ="; c.show(); cout << endl;
d.show();
d *= 2;                                             // operator*=(d,Rational(2))
cout << " *= " << 2 << " ="; d.show(); cout << endl;
if (a < 5) cout << " a < 5\n";                      // operator<(a,Rational(5));
if (1 < b) cout << " 1 < b\n";                      // operator<(Rational(1),b);
if (1 < 5) cout << " 1 < 5\n";                      // built-in inequality operator
if (d * b - a == c - 1) cout << " d*b-a == c-1 ==";
(c - 1).show();  cout << endl;
return 0;
}
```

```
5 + 1/4 = 21/4
1 - 3/2 = -1/2
7 * 1/4 = 7/4
2 / 3/2 = 4/3
7/4 += 3 = 19/4
4/3 *= 2 = 8/3
a < 5
1 < b
1 < 5
d*b-a == c-1 == 15/4
```

图10-7　程序10-7的输出结果

很多程序员争辩说，应该将运算符实现为成员函数而不是使用友元函数。不使用友元函数的主要原因是友元破坏了数据封装、信息隐藏和其他许多面向对象程序设计的好处。

的确，滥用友元函数会增加代码的复杂性，使代码难以维护。这是毫无疑问的。但是如果是合理地使用友元函数呢？什么样的友元函数是合理的？什么又是多余的呢？

要回答这个问题，我们最好来回顾一下使用C++类的主要目的。之所以要使用类，是因为如果用独立的全局函数访问数据结构，函数和数据之间的联系只体现在设计人员的思想里，而不一定会让维护人员和客户端代码程序员获悉。而且，数据封装也不是强制性的，任何函数都可以直接访问数据，而不需要访问函数。另外，我们也想有局部的类作用域，这样用于程序的一部分的函数名和数据名不会和用于程序另一部分的命名相冲突。还记得我们列举的C++类的目标表吗？希望大家能够随时将这个列表应用于对C++代码质量的评估。

有了这些评估标准，我们再来看看程序10-6的设计，它没有将重载运算符函数实现为成员函数。

```
class Rational {
  long nmr, dnm;                                    // private data
  void normalize();                                 // private member function
public:
  Rational(long n=0, long d=1)                      // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;  // const target
```

```
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
void operator += (const Rational &x);           // target changes
void operator -= (const Rational &x);
void operator *= (const Rational &x);
void operator /= (const Rational &x);
bool operator == (const Rational &other) const; // const target
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  } ;                                           // end of class specification
```

这样，数据和函数之间的联系就清晰了吗？是的，类作用域的开花括号和闭花括号表示了这种联系。那么数据是否受到保护而不能被除成员函数以外的函数访问呢？是的，数据成员被定义为private，因此不能从类外部直接访问。还存在类Rational的成员名和其他类的成员名发生冲突的危险吗？因为没有任何其他类会定义像operator+()这样的函数名，所以也没有冲突。

这个设计看起来很完美。我们再把它和使用友元函数的程序10-7对比一下。

```
class Rational {
  long nmr, dnm;                    // private data
  void normalize();                 // private member function
public:
Rational(long n=0, long d=1)        // general, conversion, default
  { nmr = n;  dnm = d;
    this->normalize(); }
friend Rational operator + (const Rational &x, const Rational &y);
friend Rational operator - (const Rational &x, const Rational &y);
friend Rational operator * (const Rational &x, const Rational &y);
friend Rational operator / (const Rational &x, const Rational &y);
friend void operator += (Rational &x, const Rational &y);
friend void operator -= (Rational &x, const Rational &y);
friend void operator *= (Rational &x, const Rational &y);
friend void operator /= (Rational &x, const Rational &y);
friend bool operator == (const Rational &x, const Rational &y);
friend bool operator < (const Rational &x, const Rational &y);
friend bool operator > (const Rational &x, const Rational &y);
void show() const;
  } ;                               // end of class specification
```

这里也有与数据相关的函数列表，就在类作用域的开花括号和闭花括号之间。对类的设计人员、客户端代码程序员和维护人员来说，这个列表都很清晰。数据是否受到了保护而不能被类括号中定义的函数以外的函数访问呢？是的，数据被声明为private，任何需要访问数据成员的函数都必须作为成员函数或者友元函数在类的括号内声明。名字冲突呢？假设我们想把重载运算符函数operator+()实现为Complex类的友元，这个函数名会和类Rational的友元函数operator+()相冲突吗？不会。处理Complex对象的operator+()函数有不同的标识：

```
Complex operator + (const Complex &x, const Complex &y);
```

因此，凭什么说友元函数破坏了数据封装、信息隐藏和其他一些面向对象程序设计的优点呢？使用友元函数的设计和使用成员函数的设计一样好。究竟使用何种方法只是个人风格问

题。本书认为，友元运算符更加容易编码和验证。另外一个重要不同是，全局运算符支持所有形式的表达式，而成员函数只支持左操作数为对象而不是数值类型值的这种形式的表达式。

提示　实现重载运算符函数时要毫不犹豫地使用友元函数。它们比成员函数更易于编写，而且支持客户代码中所有三种形式的表达式（两个操作数都是对象、只有左操作数是对象、只有右操作数是对象）。如果会让代码难以理解，就不要使用友元函数。

10.7　小结

这一章我们学习了C++的一个重要特性：重载运算符函数。与我们在前面几章中讨论的C++特性不同，重载运算符函数并不是编写高质量C++代码的绝对必要条件。

有人甚至会认为，除了像Rational和Complex这样的一小部分类以外，使用重载运算符函数会把代码弄得更复杂和难以理解。因为大多数类都不像数值类型，对它们执行数值类型的运算符并不直观。

例如，运算符函数operator+()和operator<()对类Employee或者类Transaction而言又代表什么意义呢？当然我们可以为这些运算符指定一些含义，但是这些含义并不直观和通用。如果我们将函数命名为giveRaise()和hasSeniority()，或者其他合适的名字，可能会更好。

尽管如此，运算符重载的使用仍然很普遍，特别是在C++的函数库里，如标准模板库。我们必须知道它们是干什么的，以及代码是如何实现的。

我们对成员函数和友元函数进行的对比也很重要。很多时候我们都武断地做出设计决定，而不是从面向对象程序设计的目的的角度出发进行分析。

一定不要把友元函数看成洪水猛兽。如果友元函数能够为更好的程序实现提供更多的灵活性，就应该使用它们。但不要滥用友元函数。

第11章 构造函数与析构函数：潜在的问题

重载运算符函数给面向对象程序设计提供了一种新观点，这样我们可以不再关心数据、操作及相关概念的绑定，而是将注意力集中到C++程序中内部数据类型与程序员定义数据类型的问题上。

本章是前一章内容的继续。在第10章中讨论了与数值类设计的有关问题，如Complex类与Rational类。这些类的对象正是它们自己的对象实例。与对象处理相关的所有问题都对它们适用。这些问题包括：类的声明、类成员的访问控制、成员函数的设计、对象定义、对象初始化及传递给对象的消息等。

较早所讨论的程序员定义数据类型在本质上是数值型的，尽管它们的内部结构比整数和浮点数要复杂，但我们仍能够像使用整数和浮点数一样在客户代码中对它们进行加、乘、比较等处理。

注意，上述段中的第一个“它们”指的是程序员定义的数据类型，而“整数和浮点数”指的是什么呢？既然我们将整数和浮点数与程序员定义的数据类型相比较，这里指的是整数和浮点数类型，或者说是C++中的内部数据类型，而不是整数和浮点数变量。第二个“它们”指的是什么呢？是与第一个“它们”一样的程序员定义的数据类型吗？不是，因为我们讨论的是在客户代码对它们进行的处理！但客户代码并不能处理程序员定义的数据类型，而是处理程序员定义类型的对象，对对象实例或变量进行乘、比较等等操作。之所以这样强调是想让大家在学习了大量的有关类与对象的知识后，在讨论面向对象技术时要注意这种松散语言的敏感性，并尽可能地避免产生误解。

换言之，程序员定义的数值类的对象可以像内部数据类型的变量那样被客户代码处理。这就是为什么程序员定义数据类型支持运算符重载的关键所在。处理内部数据类型和程序员定义数据类的C++原理一样，都能良好地工作。在本章中，我们将讨论其对象不能进行加、减、乘、除等操作的那些类的运算符重载问题。例如，类String被设计用来管理内存中的文本，因为其所具有的非数值特性，使这些类的运算符重载机制看起来像人为的。例如可以用重载的“+”运算符来将String串接起来或用重载的“=”运算符来对String进行比较，但对String对象的乘或除却很难处理。尽管如此，非数值类的运算符重载机制被广泛使用，大家应该掌握。

非数值类的一个重要差别是同一个类的对象所使用的数据量是可变的，而数值类的对象总是使用固定的内存。例如在Rational类中就始终有两个数据成员：一个是分子，一个是分母。

然而，在类String中，一个对象所存储的文本数量可能与另一个对象所存储的文本数量不同。如果为每一个对象都预留有相同的（足够大的）内存，这样，那个程序就会出现两个极端：内存的浪费（当实际文本数量比预留的内存少时）与内存溢出（当对象必须存储的文本数量比预留的内存多时）。这两种危险情况通常困扰着为每一个对象都分配相同数量内存的类设计者。

C++语言解决这个问题的方法是：按照类的描述为每一个对象分配一个固定大小的内存（或者是堆内存或者是栈内存），然后，如其所需那样，给堆分配它的附加内存。这个附加的堆内存大小因对象的不同而不同；即使对于同一对象，在其生命期内也可能发生改变。例如，String对象将会有附加的堆内存来容纳当前被串接到该对象的文本。

堆内存的动态管理必然导致使用构造函数和析构函数。不恰当地使用构造函数和析构函数将会影响程序性能，更糟的是这种使用可能会导致内存的混乱和程序完整性的破坏，而这些问题除了C++以外都不能被其他的语言所识别。每个C++程序员都应该意识到这些危险。因此本章虽然是继续讨论运算符函数重载机制，但我们仍在本章标题中包括了这些问题。

为了简化讨论，在第10章中我们已看到关于固定大小的Rational类的一些必要概念。本章中我们将把这些概念应用到需要进行动态堆内存管理的String类上。我们就可以琢磨该客户代码中关于各对象实例之间的编码的直观关系。我们将看到，尽管试图同等对待它们，这些对象之间的关系仍然不同于C++语言中内部数据类型的变量之间的关系。也就是说，我们会大吃一惊。

一定不要跳过本章的内容，因为在C++语言中，与构造函数和析构函数相关的危险是真实存在的。而且，我们应该懂得如何保护自己、我们的老板以及使用我们程序代码的用户。

11.1　对按值传递对象的深入讨论

在第7章中，我们对用值作为参数或用指针作为参数将对象传递给函数持有异议，而提倡使用引用来传递参数。

我们认为，按引用传递与按值传递一样简单，但前者更快，因为函数没有修改输入参数。按引用传递与按指针传递一样快，但前者的语法更简单，因为函数在执行过程中修改了输出参数。

我们也要注意到，对于函数的输入和输出参数来说，按引用传递的语法是完全一样的。因此建议在输入参数前使用const修饰符，以此表明在函数执行之后该参数没有改变。如果不使用这个修饰符，那么就应该是表明这个参数会在函数执行过程中发生改变。

我们也反对从函数返回对象值这种方式，除非是为了将其他消息发送给返回对象（表达式中的链式语法）。

按值传递这种使用方式应该局限于以下方面：用内部数据类型作为函数的输入参数，以及用内部数据类型作为函数的返回值。为什么对于内部数据类型的输入值，可以接受这种按值传递的方式呢？按指针传递会增加复杂性，容易使读者误以为在函数内部该参数改变了。按引用传递参数（带有const修饰符）并不太困难，但是稍增加了一点复杂性。由于这样的参数规模小，按引用传递它们并没有什么性能优势。这就是为什么传递参数最简单的方法（即按值传递）对内部数据类型来说是合适的。

在上一章，我们已经充分学习了编程技巧，不仅知道了各种参数传递方式的优缺点，而且也了解了它们真正的调用序列。

我们也曾多次提到过初始化和赋值，尽管它们使用相同的符号，但处理是不同的，本节我们将通过调试代码来说明其区别。

我们用程序11-1来说明上述的两个问题，该程序简化并修改了上一章中的Rational类和它的测试驱动程序。

程序11-1 按值传递对象参数的例子

```
#include <iostream.h>

class Rational {
  long nmr, dnm;                                  // private data
  void normalize();                               // private member function
public:
Rational(long n=0, long d=1)                      // general, conversion, default
  { nmr = n; dnm = d;
    this->normalize();
    cout << " created: " << nmr << " " << dnm << endl; }
Rational(const Rational &r)                       // copy constructor
  { nmr = r.nmr; dnm = r.dnm;
    cout << " copied: " << nmr << " " << dnm << endl; }
void operator = (const Rational &r)               // assignment operator
  { nmr = r.nmr; dnm = r.dnm;
    cout << " assigned: " << nmr << " " << dnm << endl; }
~Rational()                                       // destructor
  { cout << " destroyed: " << nmr << " " << dnm << endl; }
friend Rational operator + (const Rational x, const Rational y);
void show() const;
  } ;                                             // end of class specification

void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

void Rational::normalize()                        // private member function
{ if (nmr == 0) { dnm = 1; return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = -nmr; }         // make both positive
  if (dnm < 0) { sign = -sign; dnm = -dnm; }
  long gcd = nmr, value = dnm;                    // greatest common divisor
  while (value != gcd) {                          // stop when the GCD is found
    if (gcd > value)
       gcd = gcd - value;                         // subtract smaller from greater
    else value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd;          // make dnm positive

Rational operator + (const Rational x, const Rational y)
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }

int main()
{ Rational a(1,4), b(3,2), c;
  cout << endl;
  c = a + b;
  a.show(); cout << " +"; b.show(); cout << " ="; c.show();
  cout << endl << endl;
  return 0;
  }
```

在Rational类的所有函数中，我们只留下了normalize()、show()和operator+()函数。注意重载运算符函数operator+()不是类Rational的成员函数，而是它的友元函数。正因为如此，我们在这段的开头才谨慎地说是“Rational类的所有函数”，而不是“Rational类的所有成员函数”。这种提法的目的是有意强调无论从哪点看友元函数都是类的成员函数。友元函数与其他成员函数一样，在同一文件中实现，且对类的私

有成员具有同样的访问权限。但如果与Rational类以外的其他类的对象一起工作时，友元函数将变得毫无意义。它只是在调用语法上与成员函数不同，但对于重载的运算符，其语法与成员函数、友元函数是相同的。

在一般Rational类的构造函数中，我们加入了调试打印语句。此语句在每次创建并初始化Rational对象时执行，即在main()的开始处和operator+()函数中执行。

```
Rational::Rational(long n=0, long d=1)              // default values
  { nmr = n; dnm = d;                               // initialize data
    this->normalize();
    cout << " created: " << nmr << " " << dnm << endl; }
```

我们还增加了一个有调试打印语句的拷贝构造函数，此语句在一个Rational类对象被另一个Rational对象的数据成员初始化时执行。例如，当按值传递参数给operator+()函数时或从operator+()函数返回一个Rational对象时，此语句将执行。

```
Rational::Rational(const Rational &r)       // copy constructor
  { nmr = r.nmr; dnm = r.dnm;               // copy data members
    cout << " copied: " << nmr << " " << dnm << endl; }
```

当Rational参数按值传递给友元运算符函数operator+()时，将调用这个构造函数。而在operator+()函数返回对象值时，并不调用该拷贝构造函数。因为在operator+()函数返回值之前，已优先调用了带有两个参数的一般构造函数。

在Rational类中，析构函数没有什么实际意义的工作。我们之所以保留它，是因为当撤销一个Rational对象时，将执行调试语句。

这里最有意义的函数是重载的赋值运算符函数。其作用是将一个Rational对象的数据成员拷贝给另一个Rational对象的数据成员。那么它与拷贝构造函数有何不同呢？至少在这里，它们没有什么区别。其实它们的返回值类型是不同的——拷贝构造函数一定没有返回值，而赋值运算符与其他大多数成员函数一样必须有一个返回值类型。只是为了简便，在这里返回值类型为void。

```
void Rational::operator = (const Rational &r)     // assignment
  { nmr = r.nmr; dnm = r.dnm;                     // copy data
    cout << " assigned: " << nmr << " " << dnm << endl; }
```

该重载的赋值运算符是一个二元运算符。我们是如何知道这一点的呢？首先，它有一个Rational类类型的参数，而且它是一个成员函数，而不是友元。正如任何一个带有一个参数的成员函数那样，它对两个对象进行操作：一个是消息传递目标，另一个是参数。其次，其语法上使用赋值作为运算符。二元运算符总是放在第一个操作对象和第二个操作对象之间。当两个操作对象相加时，先写第一个操作对象，再写运算符和第二个操作对象（如：a + b）。使用赋值运算符时，也是如此（即先写第一个操作对象，再写运算符和第二个操作对象，如a = b）。在函数调用的语法中，对象a是消息传递的目标：在上面的赋值运算函数中，nmr和dnm都属于目标对象a；对象b是此函数调用的参数：在上面的赋值运算符中，r.nmr和r.dnm都属于实际变量b。因此赋值运算符的函数调用语法为：a.operator=(b)。

由于该运算符返回的是void空值，因而它不支持客户代码中的链式赋值，如：a =b =c。这个表达式会被编译程序解释为a = (b = c)，即b = c（或b.operator=(c)）的返回值将作为赋值a.operator=(b.operator=(c))的参数。只有当赋值运算符返回类类型

（这里是Rational）的值时，这个表达式才有效。由于赋值运算符返回的是空值void，所以编译程序认为这个链式表达式有语法错误而予以标注，初看起来，这里的赋值运算符并不重要，在本章稍后我们将会用到链式赋值。

程序11-1的输出如图11-1所示。前面三个“created”消息是main()中三个Rational对象的创建和初始化所产生的。两个“copied”消息是在将数据流传递给重载运算符函数operator+()时产生的。紧接着的“created”消息是来自operator+()函数体内调用的Rational构造函数。

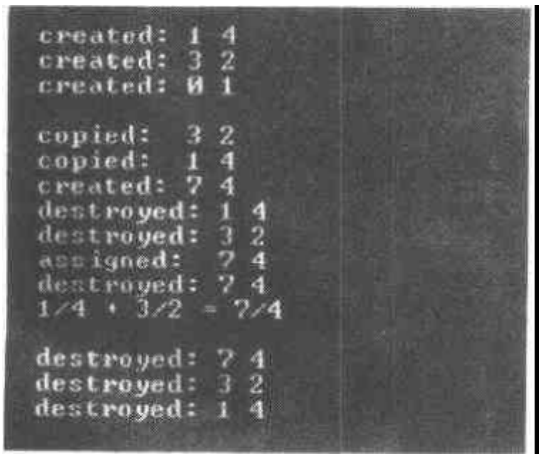


图11-1　程序11-1的输出结果

所有这些构造函数的调用在其函数执行的开始发生。接着出现一系列的事件，当执行到函数体结尾的闭花括号处时，将撤销局部对象与临时对象。当两个实际参数（3/2和1/4）的局部拷贝被撤销时便产生前两个“destroyed”消息，此时，为这两个对象调用析构函数。含有参数总和的对象在赋值运算符使用它之前不能撤销，接着的“assigned”消息来自于调用重载赋值运算符，“destroyed”消息来自于在operator+()函数体中所创建对象的析构函数。当执行到达main()结尾的闭花括号处时，调用各析构函数，产生了最后三个“destroyed”消息，同时撤销对象a、b、c。另外，由于并未调用拷贝构造函数，所以在输出中没有出现“copied”消息。

如果在operator+()函数接口中加入两个“&”符号，这个事件序列将有不同意义。

```
Rational operator + (const Rational &x, const Rational &y)          // references
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }
```

在C++编程中对不同部分程序代码的一致性要求是很难满足的。这里，我们改变函数原型的界面，并修改了类规格说明中的函数声明。（再次说明，它是成员函数还是友元函数并不重要。）在这种情况下，不保持程序代码相关部分的一致性并不是致命错误——因为编译程序会警告该程序代码有语法错误。

加入了operator+()函数的程序11-1的执行结果如图11-2所示。我们看到少了4个函数调用：没有创建两个参数对象，而且也没有撤销两个参数对象。

提示　避免将对象实例作为值参数来进行传递，这将导致不必要的函数调用。建议按引用来传递参数。如果可能，在函数接口中将它们标注为常量对象。

下面，我们用示例来说明初始化与赋值之间的区别。在程序11-1的表达式c = a + b中，

对变量c赋值。那么我们怎么知道它是赋值还是初始化？由于在c的左边没有类型名，其类型在main()的开始处早已定义，所以是赋值，与此不同，下面这个main()版本中，创建对象c并立即将它初始化为a与b的和，而不是在创建后再由单独的语句对c进行赋值

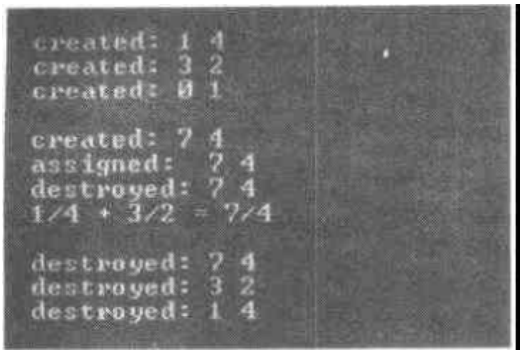


图11-2　程序11-1按引用传递参数时的输出结果

```
int main()
{ Rational a(1,4), b(3,2), c = a + b;
  a.show(); cout << " +"; b.show(); cout << " ="; c.show();
  cout << endl << endl;
  return 0; }
```

如果程序11-1中的主函数是上面这个main()函数，且按引用传递参数，其执行结果将如图11-3所示。我们看到这里没有调用赋值运算符，也没有调用拷贝构造函数——这是按引用来传递参数而不是按值传递参数的自然结果。

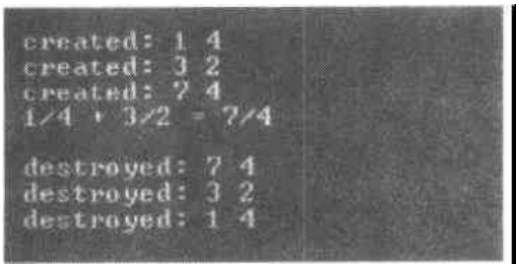


图11-3　程序11-1按引用传递参数并使用对象初始化而不是赋值的输出结果

稍后，我们将用类似的技术来示例说明String类的初始化和赋值两者之间的不同。

提示　要区别对象初始化和对象赋值的不同　初始化时，将调用构造函数而不是调用赋值运算符。在赋值时，将调用赋值运算符而不是构造函数

避免按值传递对象参数，并区别对象初始化和对象赋值的不同是非常重要的概念。要做到在阅读客户代码时，能清楚地分辨出“哪里调用了构造函数、哪里调用了赋值运算符”。要注意锻炼自己分析这类问题的直觉能力。

11.2　非数值类的运算符重载

正如我们在本章简介中所提到的那样，将数值类上的基本运算符进行扩展是很自然的这些类的重载运算符函数与基本运算符大致相同。客户端代码程序员及维护人员不可能对此产生误解。可以同样地对待内部数据类型与程序员定义类型，这是一个很好的思想，实现起

来也很简单。

运算符也可以运用到非数值类的对象上，但是要对加、减或其他运算符的意义进行扩展。正如命令输入的图标在图形用户界面中所表示的意义一样。

最初只有命令行界面，用户不得不输入长长的带有参数、关键字、转换符等的命令。后来出现了有文本输入条目的菜单，用户通过选择菜单的条目来输入所需命令，而不再需要输入整个命令行。接着有了热键，用户只需要按一下热键组合就可以直接激活命令，而不需要在键盘上动手指并在几个菜单与子菜单之间进行选择。之后又出现了带有命令按钮的工具栏，用户只需要点击一下工具栏就可以激活命令，而不需要知道热键。而且这些命令按钮上的图标非常直观清晰，如：Open、Close、Cut、Print。当加入了越来越多的图标如：New、Paste、Output、Execute、Go等后，就越来越不那么直观了。

为了帮助用户学习所加入的各个图标所代表的操作，系统中加入了工具条提示信息。这样，用户界面变得更加复杂；应用程序也需要更多的磁盘空间、内存和编程精力，现在用户也许还没有当初使用习惯的菜单和热键的感觉好。类似地，我们最初只有数值类的运算符重载机制，而现在将要使用非数值类的运算符函数。这就要求掌握更多的规则、写更多的代码，处理更多的复杂性问题。在客户代码中使用传统的函数调用而不使用现代的重载运算符可能会更好一点。

11.2.1 String类

我们来讨论一个比较流行的非数值类使用的重载运算符函数的例子：对于文本串接使用加法运算符。

我们考虑一个String类，该类拥有两个数据成员：一个是指针，指向动态分配的字符数组；另一个是整数，表示可被插入到动态分配的堆内存中的最大有效字符数。实际上，C++标准库中包含有String类（其第一个字母是小写），此类的设计用来满足绝大部分文本操纵的需求，这是一个很有用的类，比我们在这里将要讨论的类强有力得多。但由于其复杂性，在这些例子里我们不能使用这个类。我们讨论的细节是动态内存管理及其结果。

在客户代码中有两种方式来创建类对象：通过指定最大有效字符数或通过指定字符串的文本内容。指定字符数需要一个整数参数；指定文本内容也需要一个字符数组参数。由于这些参数的类型不同，因而需要使用不同的构造函数。由于每个构造函数恰好只有一个非类类型的参数，要把这些参数转换为一个类的值，故将这些构造函数称为转换构造函数。

第一个转换构造函数，其参数表示分配给字符串的长度，缺省参数值为0。使用这个缺省值（没有指定参数）创建一个String对象时，分配给该对象的文本长度就为0。在这种情况下，第一个转换构造函数用作一个缺省构造函数（如String s;）。

第二个转换构造函数，其参数是字符数组，它没有缺省参数值。也不难给它一个缺省值，即一个空字符串。但编译程序将很难解释函数调用String s;。我们是调用缺省值为0的第一个转换构造函数，还是调用缺省值为空字符串的第二个构造函数呢？

在客户代码中调用成员函数modify()可修改字符串的当前内容，该modify()成员函数指定了目标对象的新文本内容。要访问String对象的文本内容，要使用show()函数。该函数将指向分配的堆内存的指针返回给对象。在客户代码中可使用这个指针打印字符串的内容，或将其内容与其他文本进行比较等。程序11-2描述了String类的实现。

程序11-2　动态分配堆内存的String类

```
#include <iostream>
using namespace std;

class String {
  char *str;                          // dynamically allocated char array
  int len;
public:
  String (int length=0);              // conversion/default constructor
  String(const char*);                // conversion constructor
  ~String ();                         // deallocate dynamic memory
  void modify(const char*);           // change the array contents
  char* show() const;                 // return a pointer to the array
} ;

String::String(int length)
{ len = length;
  str = new char[len+1];              // default size is 1
  if (str==NULL) exit(1);             // test for success
  str[0] = 0; }                       // empty String of 0 length is ok

String::String(const char* s)
{ len = strlen(s);                    // measure length of incoming text
  str = new char[len+1];              // allocate enough heap space
  if (str==NULL) exit(1);             // test for success
  strcpy(str,s); }                    // copy text into new heap memory

String::~String()
{ delete str; }                       // return heap memory (not the pointer!

void String::modify(const char a[])   // no memory management here
{ strncpy(str,a,len-1);               // protect from overflow
  str[len-1] = 0; }                   // terminate String properly

char* String::show() const            // not a good practice, but ok
{ return str; }
int main()
{
  String u("This is a test.");
  String v("Nothing can go wrong.");
  cout << " u = " << u.show() << endl;        // result is ok
  cout << " v = " << v.show() << endl;        // result is ok
  v.modify("Let us hope for the best.");      // input is truncated
  cout << " v = " << v.show() << endl;
  strcpy(v.show(),"Hi there");                // bad practice
  cout << " v = " << v.show() << endl;
  return 0;
  }
```

11.2.2　堆内存的动态管理

第一个转换构造函数的第一行代码设置了数据成员len的值；第二行代码通过分配所需的堆内存量设置了数据成员str的值。接着测试内存分配成功与否，并将所分配内存的头部清空（即设为字符串'\0'）。对于任何C++库函数，该文本内容为空，尽管客户代码已指定了字符数的空间。

如果在客户代码中定义了一个String对象而没有提供参数，该构造函数用作一个缺省构造函数，它将在堆中分配一个字符并设为'\0'表示空字符串。

对于下列语句，这个构造函数的每条语句执行时的内存图如图11-4所示。

```
String t(20);                    // 21 characters on the heap
```

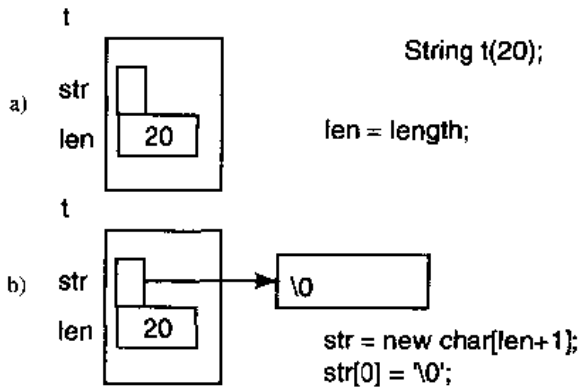


图11-4 程序11-2中第一个转换构造函数的内存图示

图11-4a表示了构造的第一个阶段，图11-4b表示了构造的第二个阶段。图中的矩形表示String对象t有两个数据成员：指针str和整数len。这些数据成员可能占有相同数量的内存，我们将表示指针的矩形画得小一点儿是为了强调它并不包含可计算的数据这个事实。对象名t、数据成员名str和len写在表示对象的矩形框之外。

图11-4a表示执行语句len = length之后，数据成员len被初始化为20（它包含了一个值），而指针str保持未初始化状态（它想指到哪里就指到哪里）。图11-4b表示执行构造函数体的其他语句之后，分配了堆空间（21个字符），并由指针str指向该空间，其第一个字符设置为0。对于一个简单的对象画一个图看上去似乎过于麻烦，但我们建议为所有处理指针和堆内存的代码画这些图示。这是培养有关动态内存管理的编程直觉的最好办法。

第二个转换构造函数的第一行代码计算出客户代码中所指定的字符串长度，并将该长度值赋给数据成员len。第二行代码通过分配str所指向的所需的堆内存，对数据成员str进行了设置，并将程序所指定的字符拷贝到所分配的内存中。库函数strcpy()从参数数组中拷贝字符，并在其末尾添加终止符0。

图11-5表示了下面语句的对象初始化步骤。

```
String u("This is a test.");          // 15 symbols, 16 characters on the heap
```

有三种方法处理那些保存堆内存大小的数据成员：第一种方法是在数据成员中保存总共分配的堆内存大小（可容纳的符号数加1）；第二种方法是用一个数据成员保存有用的符号数，并在堆内存中分配了字符后将该数目加1。本书使用了第二种方法，但很难解释为什么这种方法比第一种更好。不过，我们并不愿倒过来，因为同样很难理解为什么第一种方法是较好的。

第三种方法是根本不保存字符串长度作为一个数据成员，而是通过调用strlen()函数来即时地计算其长度。这是一个时间与空间折中的例子。如果我们不经常需要其长度，并且不喜欢为每一个String对象额外分配一个整数，那么第三种方法较好。

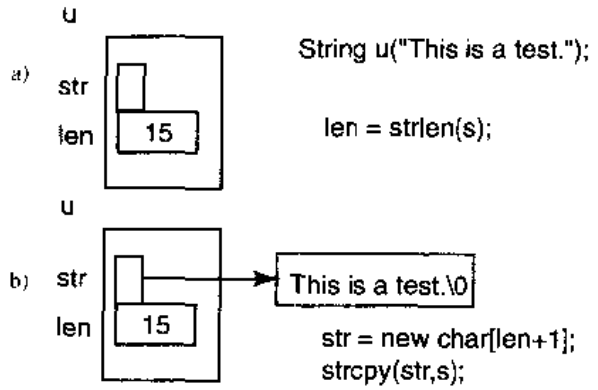


图11-5　程序11-2中第二个转换构造函数的内存图

由于每一个String对象要分别分配堆内存空间，许多程序员都感到该堆内存应当看作对象的一部分。根据这样的观点，在客户代码中String对象长度是变长的，其长度依赖于所分配的堆内存大小。这个观点是可行的，但其结果将导致更难于解释构造函数和析构函数的工作机制，在某种程度上模糊了类本身的概念。

我们推荐使用图11-4和图11-5中所演示的方法，它反映了C++的基本原则，即类是对象实例的蓝图。该蓝图对于所有的String对象都是相同的。根据这个蓝图，每个String对象有两个数据成员，而且每个String对象的大小也是相同的。当执行客户代码中的下列这条语句后，在栈中为对象t分配两个数据成员，为某个特定对象执行的String成员函数分配堆内存。不同的String对象可以有不同数量的堆内存，或者它们也可以释放内存，或者在不改变其标识的情况下要求更多的内存。

```
String t(20);          // two data members are allocated on the stack
```

当String对象本身是在堆中分配时，这种方法不会改变。我们来讨论这个客户代码例子。

```
String *p;                    // no String object, pointer is created on the heap
p = new String ("Hi!");       // two data members plus 4 characters on the heap
```

这里，一个未命名的String对象（由指针p指向它）获得了一个整数和一个堆中的字符指针。当该对象被创建后，构造函数将在堆中另外分配4个字符，并将指针str指向所分配的内存。

这种方法让我们很容易地想到同一个类的所有对象的大小是相同的。当创建一个对象时有两个独立的过程：对象的创建（大小总是相同的）和一个构造函数的调用。这将会初始化对象的数据成员，包括指向堆内存的指针。

析构函数删除堆中动态分配的内存，就在对象即将要撤销时调用析构函数。当撤销对象时，分配给其数据成员str和len的内存也要撤销，并还回供以后使用。如果对象是在栈中分配的，如程序11-2main()中的对象u和v，该内存将退回给该栈。如果对象是在堆中分配的（如由指针p指向的未命名的对象），分配给len和str的内存将还回到堆中。但在任何情况下，在数据成员len和str消失之前，析构函数所删除的内存（由指针str指向的）都要还回到堆中。否则，析构函数中的语句delete str;将是不合法的。

modify()函数改变动态分配的堆内存内容，它使用库函数strncpy()确保内存不会出现讹用，即使客户代码错误地提供某字符串，且该字符串的长度超过了为对象所分配的动

态内存的大小。在溢出情况发生时，strncpy()不会用空终结符终止字符串。这正是我们在函数的末尾那样处理的原因所在。当新的字符串长度比有效内存短时，这样处理似乎有些多余。但要记住，在这种情况下strncpy()将用0来填充字符串中其余的部分，多做一次这样的处理不会降低程序的执行速度。

modify()函数不能对字符串的初始长度进行扩展，大多数String设计不允许程序员改变String对象的内容。在这种情况下，程序员可以为所需的不同内容创建和使用另一个对象。我们实现了这种折中办法。完善的修改功能需要编写更多的代码，而且还要涉及到许多其他额外问题的讨论。这个小的modify()函数对于我们这里所讨论的问题而言已经足够了。

show()函数返回指向动态分配内存的指针。在程序11-2的main()客户代码中用示例说明了此函数的两种用法。第一种用法是打印show()消息的目标String对象的内容。第二种用法用show()函数的返回值作为客户代码中调用strcpy()的输出参数，以修改对象的内容。第一种用法是合法的，第二种用法比较鲁莽，它会使维护人员感到害怕而不易理解代码开发人员的意图。

高级计算机语言之一的APL（A Programming Language）非常复杂，并仍在使用，主要用于编写金融应用软件。这种语言的字符集非常庞大，以至于需要一个特殊的键盘。但它有强大的数组和矩阵处理功能。APL程序员很喜欢这种语言，认为用APL写一些代码并给朋友看让他们猜其意义是一种很好的品味。

我们并不是暗示有这样一种精神状态的程序员会被解雇。但当其他人必须维护这些程序员的代码时，这些程序员不应参与这样的工程项目。如今，如果一个程序员编写的代码不易让人们读懂，需要花过多精力去理解它，这样的事情不值得炫耀。

```
strcpy(v.show(),"Hi there");                    // bad practice
```

注意，让我们气愤的主要是这样的事实：程序维护人员不得不花费额外的精力去读懂这些代码。至于那些并不计算对象的有效堆内存大小而又可能因此而导致耗用内存的代码存在重大的问题，它们只是雪上加霜而已。在客户代码和服务器String类之间采用不同的职责可以修正上述问题。

```
int length = strlen(v.show());              // get available space
strncpy(v.show(),"Hi there",length);        // pushes responsibility up
```

对于第二个转换构造函数创建的String对象，length的值是总共的有效空间。对于第一个转换构造函数创建的对象，length的值是所存储的最后一个字符串的长度，此长度比总共的有效空间要小。更重要的是，这种方法违背了将职责从客户推到服务器。并对客户代码隐藏数据操纵细节的基本原则。

在这里，客户代码负责低层次数据操纵，即使在该代码中并未使用String数据成员的名字。如果想要保护堆数据不被破坏，那么应在服务器代码中包含动态存储器的有效空间的检查。一个比较好的办法是使用服务器函数的名字而不是直接对服务器数据进行处理，并将保护堆内存的任务推给服务器。在这里，该方法安全有效，不再赘述。

```
v.modify("Hi there");          // it tests for available space
```

图11-6给出了程序11-2的输出结果，说明了调用函数modify()通过截短客户代码数据防止动态内存的溢出。

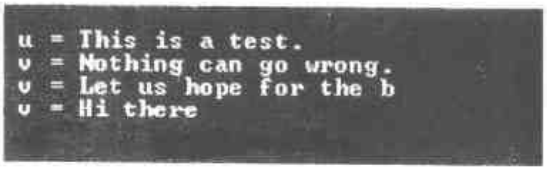


图11-6　程序11-2的输出结果

在这里并没有保护函数show()所返回的指针的使用，下面是函数String::show()可能导致的内存讹用的例子。

```
char *ptr = v.show();          // reckless practice
ptr[200] = 'A';                // memory corruption
```

或者，如果喜欢使用对象的链式标注，可以只用下面的一个语句来处理。

```
v.show()[200] = 'A';           // reckless practice, memory corruption
```

这不是一种好的用法。

11.2.3　保护客户代码中的对象堆数据

C++提供了一种保护客户代码中对象堆数据的方法，那就是使用由成员函数返回的指针。将指针定义为指向一个常量可以防止滥用。例如，将函数show()的返回值定义为指向一个常量字符的指针，而不是定义为指向一个非常量字符的指针，正如我们在程序11-2中所做的处理一样。

```
const char* string::show( ) const       // good practice: return const
{ return str; }
```

现在，如果客户代码尝试通过成员函数show()返回的指针来修改动态内存的内容，编译程序将会把它标注为语法错误。

```
strcpy(v.show( ), "Hi there");       // error, not just bad practice
```

这样设计服务器String类，客户代码就不得不使用modify()函数改变对象的状态。结果，客户代码以调用服务器函数的方式将保护操作推向服务器类，而不是强制客户代码处理服务器设计的细节（有限的堆空间）。

11.2.4　重载的串接运算符

下一步是设计重载的运算符函数来串接两个String对象：将第二个对象的内容添加到第一个对象内容的后面。这意味着在客户代码中可以按下面的方式使用这个重载的运算符函数。

```
String u("This is a test. ");          // left operand
String v("Nothing can go wrong.");     // right operand
u += v;                                // expression: operand, operator, operand
```

执行完这段代码后，对象v的内容应保持相同，而对象u的内容将改变为："This is a test. Nothing can go wrong."。

如果我们把这个运算符函数作为一个成员函数实现，这样对象u应是消息的目标对象，对象v应是函数调用的参数。上面这段小程序的最后一个语句的实际意义是：

```
u.operator+=(v);    // meaning of u += v; -> u is the target, v is the parameter
```

因此，在该函数的界面中参数应加上const修饰符，而不是让成员函数本身加上const修饰符。返回类型为void，这将限制在链式表达式中使用该运算符，但这对于客户端代码程序员来说不是一个很严重的限制。

```
void operator += (const String s);    // concatenate parameter to target object
```

我们知道按值传递对象不是一个好方法，但在这里我们假设没有性能上的问题。毕竟，String类型的对象只有两个较小的数据成员：一个字符指针和一个整数，拷贝这些数据成员不应花费很长的时间。

String串接算法应包括以下步骤：

1) 将两个字符数组的长度相加，定义字符的总共长度。

2) 分配容纳这些字符及终止符0的堆内存。

3) 测试内存分配成功与否，如果系统内存不足则放弃操作。

4) 将目标对象的字符拷贝到新分配的堆空间中。

5) 将参数对象的字符串接到新分配的堆空间中。

6) 将目标对象的指针str指向新分配的堆空间中。

图11-7表示了上述步骤（排除了系统内存不足则放弃操作的异常情况），并用C++语句实现这些步骤。我们在客户代码中使用某些较短的字符串简化事件的跟踪。

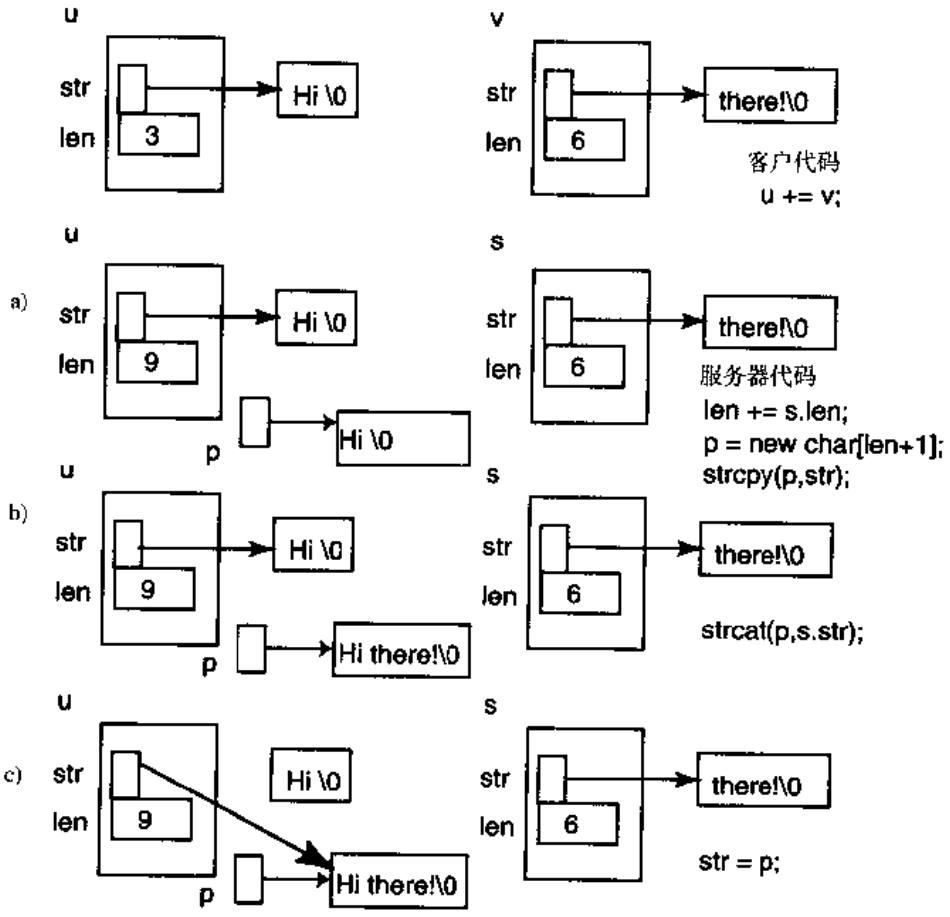


图11-7 String串接运算符函数的内存图

图11-7的最上面是两个String对象u（其内容为“Hi”）和v（其内容为“there!”）。

图11-7a表示了第一个对象的域len被修改后这两个对象的状态：堆内存已分配，对象u已有的内容被拷贝到堆内存（算法中的步骤1)~4)）。图11-7b表示执行了步骤5之后堆内存的状态。图11-7c表示当目标对象u的指针str被设置为指向新分配的堆内存后对象的状态（步骤6)）。

综上所述，我们得到以下服务器代码。

```
void String::operator += (const String s)     // object parameter
{ char* p;                                    // local pointer
  len = strlen(str) + strlen(s.str);          // total length
  p = new char[len + 1];                      // allocate heap memory
  if (p==NULL) exit(1);                       // test for success
  strcpy(p,str);                              // copy the first part of result
  strcat(p,s.str);                            // concatenate the second part
  str = p; }                                  // set str to point to new memory
```

为这个简单的算法写出如此详细的步骤，并对内存处理的每一步都画出单独的图，这样看起来有点儿多余。如果有这样的感觉还不错，但只有少数幸运的人才有这种感觉。对于大多数人而言，指针操作模糊不清，与人的直觉相抵触。

只有经验丰富的程序员才能注意到目标对象所占有的堆内存没有正确地恢复。图中很明确地显示了这一点。

我们认为画图是培养关于内存管理和发现错误的直觉能力的惟一途径。对于模糊不清的语句，与其用跟踪器和其他复杂的工具，还不如多花一点时间画一些很详细的图示来帮助理清思路。

当然，画图只是一种工具，其目的只是让我们确实理解每一个语句的意思。

11.2.5 防止内存泄漏

正如我们所提到的，图11-7显示了函数调用开始时目标指针str指向的堆字符数组没有正确地恢复。当指针str转而指向新分配内存段时（局部指针p正指向着），它将变得不可访问。这就是内存泄漏——指针操作和内存管理中经常出现的一个错误。为防止内存泄漏，在指针str转而指向新分配的数组之前，这个字符数组的空间必须还回到堆中。

```
void String::operator += (const String s)    // object parameter
{ char* p;                                   // local pointer
  len = strlen(str) + strlen(s.str);         // total length
  p = new char[len + 1];                     // allocate enough heap memory
  if (p==NULL) exit(1);                      // test for success
  strcpy(p,str);                             // copy the first part of result
  strcat(p,s.str);                           // concatenate the second part
  delete str;                                // return existing dynamic memory
  str = p; }                                 // set str to point to new memory
```

图11-8与图11-7相似，它表示由目标数据成员str所指向的堆字符数组在delete操作后消失了。只有这样之后，指针str才转而指向新的堆数组。

随着对内存泄漏的关注，我们承认在讨论重载运算符函数时，我们介绍了其真实情况，但仅仅是部分真实情况，没有介绍所有的真实情况。原因是我们想确保在面临更复杂、更危险的问题之前，关注较小的、不太难的问题。我们希望能集中注意力。

这个讨论应能提示，当编写自己的C++程序时所必须认识到的起危害作用的结构。问题的核心是：作为值参数传递对象。

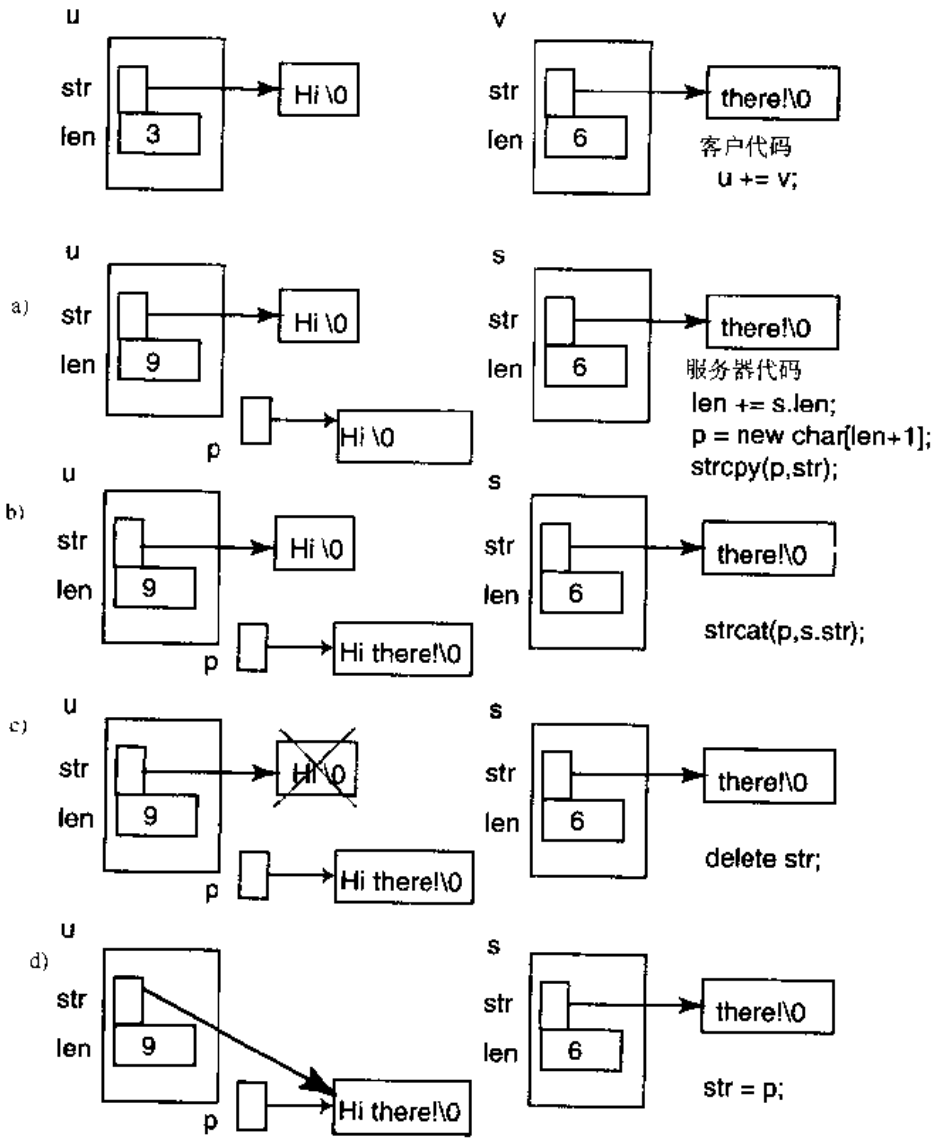


图11-8 修正的String串接运算符函数的内存图

11.2.6 保护程序的完整性

当实际参数按值传递时，无论是否是对象，它的值都要拷贝到栈的一个局部自动变量中。这个拷贝过程是按成员的顺序进行的。

这对于内部数据类型的参数来说没有任何问题；但对于像Rational和Complex这样简单的类就会有一点细微的性能上的影响；对于其对象需要大量内存的类，这将是一个很实际的性能问题。

更严重的是，如果类中有数据成员是指向动态分配的堆内存的指针时，这将是一个严重的完整性问题。让我们来看一下带有值参数的函数在函数执行的关键点情况——在函数调用的开始和函数终止时。这些操作是在函数体的开花括号和闭花括号处。

在按值传递过程中，当创建一个实际参数对象的副本时，将调用系统提供的拷贝构造函数。该构造函数将实际参数的数据成员拷贝到局部副本（即形式参数对象）的相应数据成员中。当拷贝指针数据成员str时，形式对象的指针接收到存储在实际参数对象指针中的值，

也就是为实际参数对象分配的堆内存的地址。

结果，实际参数中的指针和其局部副本中的指针都指向相同的堆内存段，而且每个对象都认为自己独占了该内存。

我们尝试将上述情形表示在图11-9中。实际上，迄今为止所介绍的这些是没有改变重载运算符函数的工作机制。正因为此，在较早我们提示说所有介绍的都是真实情况，但仅仅是部分真实情况而已。

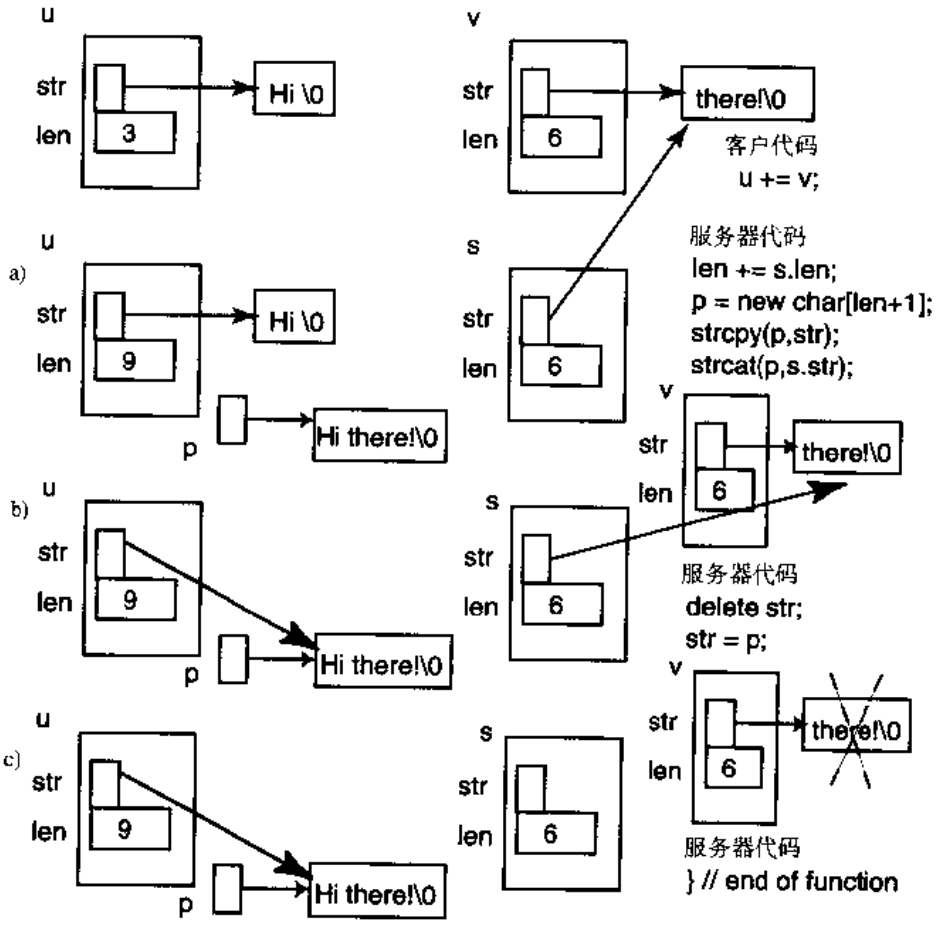


图11-9　按值传递String对象的内存图

图11-9中显示了整个真实情况，包括一个局部对象s的数据成员被初始化为实际参数v的值。图11-9a中表示了这个局部对象s[⊖]和实际参数v[⊖]共享相同的堆内存。图11-9b中表示在分配了新的堆内存并初始化、替换了目标对象中已有的堆内存后，局部对象s和实际参数u继续共享相同的堆内存段。

整个真实情况也应包括函数终止处。当函数执行到闭花括号处，函数终止，撤销局部副本对象（String s）。按通常的编程经验，这意味着对象内存（这种情况下是指针和整数）消失。但是在C++中没有这样的对象撤销。每个对象在撤销之前都要调用一个函数，即调用析构函数。

当析构函数被调用时，处理析构函数代码所应处理的事情，还回由对象指针指向的内存。

⊖ 原文为v似有错。——译者注

⊖ 原文为u似有错。——译者注

```
String::~String()
{ delete [] str; }        // return heap memory pointed to by pointer
```

图11-9c中显示在调用析构函数之后但在撤销局部对象之前，局部对象s和实际参数v的状态。它表示了局部对象和实际参数都丢失了其堆内存（删除由指针str所指向的内存）。当然，这个行为并不影响目标对象的状态，因为没有撤销目标对象。当重载运算符函数终止时，目标对象所处的状态与图11-8反映的状态完全相同。这段客户代码将产生正确的结果。

```
String u("Hi "); String v("there!");
u += v;
cout << " u = " << u.show() <<endl;        // it displays "Hi there!"
```

然而，当撤销形式参数s时由析构函数还回的内存并不属于对象s，它属于（而且仍应属于）实际参数，也就是在该客户的作用域中定义的对象v。函数调用后，被当做实际参数来进行按值传递的客户对象被剥夺了其动态分配的内存。该函数调用以后在客户代码中再使用此对象将出现错误。

```
String u("Hi "); String v("there!");
cout << " u = " << u.show() << endl;      // it displays "Hi "
cout << " v = " << v.show() << endl;      // it displays "there!"
u += v;
cout << " u = " << u.show() << endl;      // it displays "Hi there!"
cout << " v = " << v.show() << endl;      // displays what it wants
```

再次检查刚刚被打印过的对象v的值，并将它作为函数调用operator+=()的右值，这个做法看上去有些不太明智。我们这样做只是因为我们知道这个实现中有问题。显然，该对象的值必须与它在表达式u +=v中作为一个操作数时的值相同。这是常规的编程经验，在C++的大多数情况下是可行的，但并不是在所有情况下都行得通。这样，我们应尽快地培养另一种编程直觉能力。我们介绍的所有这些情况是因为在这个看起来无辜的客户代码中，对象v的文本内容可以是任意的，并且认为它仍与以前的状态相同，但使用该对象却是考虑不周而不恰当的。

当然，C++编程并不麻烦。但C++程序员必须明白像上面这个例子中那样小段的代码将会怎样地执行。

事情还没有结束，在另一个作用域的闭花括号处存在着同样的情况。经常要注意作用域括号，在那里将会发生很多事情。当客户代码执行到其作用域的闭花括号处并终止时，要为所有的局部对象调用类的析构函数，包括运气不好的对象v。v被用作函数调用的实际参数，在调用终止时剥夺了其动态内存。析构函数试图回收由对象数据成员str所指向的区域，但该内存已还回给系统。如果我们正在设计这门语言，会让它成为“没有操作”(no op)。但这不可能，因为在C++中不允许对同一指针反复使用delete操作，这是错误的用法。

不幸的是，对于上述错误的用法，编译程序并不会提示有语法错误要改正。编译程序编写者并没有责任要跟踪执行流与提示错误，它只用保证代码在语法上是正确的。上述错误的用法也不意味着程序可以编译，运行并产生重复的不正确结果。它只是简单地意味着这样尝试的结果是“没有定义的”。实际上，它们依赖于具体的平台。应用程序如何运作依赖于操作系统。系统也许会崩溃，程序可能会悄悄地不正确地运行，或者它将正确地运行到未来的某一时刻。

程序11-3是上述糟糕方案的完整程序。该程序在一些机器中运行后，输出的结果如

图11-10所示。

程序11-3　带有一个值参数的重载串接函数

```
#include <iostream>
using namespace std;

class String {
  char *str;                              // dynamically allocated char array
  int len;
public:
  String (int length=0);                  // conversion/default constructor
  String(const char*);                    // conversion constructor
  ~String ();                             // deallocate dynamic memory
  void operator += (const String);        // concatenate another object
  void modify(const char*);               // change the array contents
  const char* show() const;               // return a pointer to array
  } ;

String::String(int length)
{ len = length;
  str = new char[len+1];
  if (str==NULL) exit(1);
  str[0] = 0; }                           // empty String of zero length is ok

String::String(const char* s)
{ len = strlen(s);                        // measure length of incoming text
  str = new char[len+1];                  // allocate enough heap space
  if (str==NULL) exit(1);                 // test for success
  strcpy(str,s); }                        // copy incoming text into heap memory

String::~String()
{ delete str; }                           // return heap memory (not the pointer!)

void String::operator += (const String s) // pass by value
{ len = strlen(str) + strlen(s.str);      // total length
  char *p = new char[len + 1];            // allocate enough heap memory
  if (p==NULL) exit(1);                   // test for success
  strcpy(p,str);                          // copy the first part of result
  strcat(p,s.str);                        // add the second part of result
  delete str;                             // important step
  str = p; }                              // now p can disappear
const char* String::show() const                  // protect data from changes
{ return str; }

void String::modify(const char a[])               // no memory management here
{ strncpy(str,a,len-1);                           // protect from overflow
  str[len-1] = 0; }                               // terminate String properly

int main()
{ String u("This is a test. ");
  String v("Nothing can go wrong.");
  cout << " u = " << u.show() << endl;            // result is ok
  cout << " v = " << v.show() << endl;            // result is ok
  u += v;   // u.operator+=(v);
  cout << " u = " << u.show() << endl;            // result is ok
  cout << " v = " << v.show() << endl;            // result is not ok
  v.modify("Let us hope for the best.");          // memory corruption
```

```
    cout << " v = " << v.show() << endl;          //  ????
    return 0;
    }
```

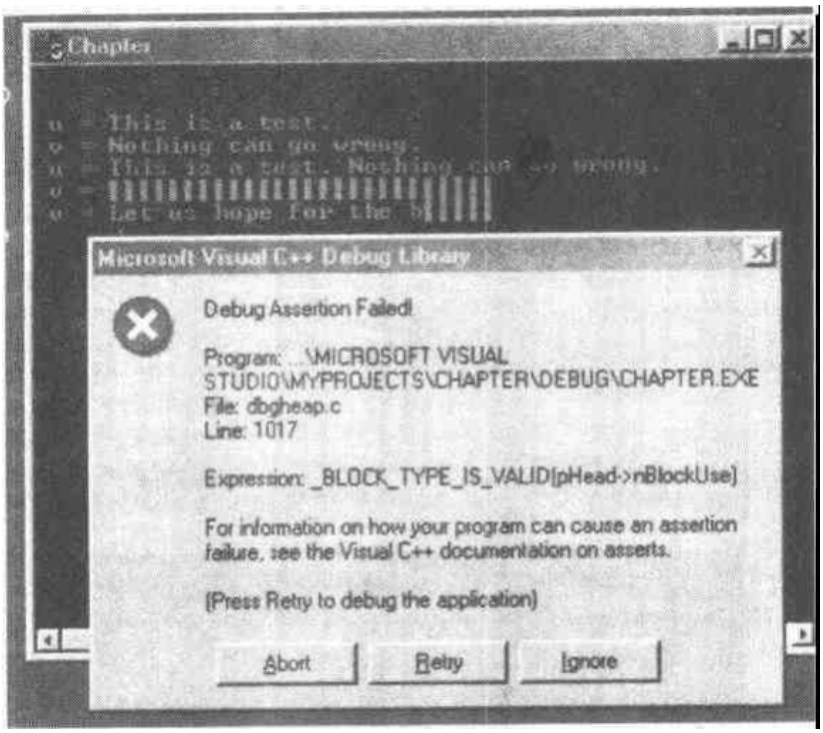


图11-10 程序11-3的输出结果

值得注意的是，所有这些糟糕的事情都是在函数终止时发生的。第一件糟糕的事情是当服务器重载函数operator+=()终止时，调用形式参数的析构函数，从而剥夺实际参数v的堆内存。第二件糟糕的事情是客户函数main()终止时，对象v已超出其作用域范围，重复删除其堆内存。

实际上，在C++中，“一个错误”指的是重复删除堆内存，而删除一个NULL指针不是错误，而是“没有操作”。于是，有些程序员试图通过在析构函数中将设置指向堆内存的指针为NULL来解决这个问题。

```
String::~String()
{ delete str;          // return heap memory
  str = 0; }           // set to null to avoid double deletion
```

这是一个好主意。但它并不按所想像地那样工作。设置为0的指针属于在几微秒内将要撤销的对象，而指向可设置为0的同一内存的指针是属于第二个对象的，它对于另一个对象析构函数的执行来说是无用的。即使它有效，也只能防止“一个错误”，而不能将不正确删除的内存重新恢复。

11.2.7 如何由此及彼

希望大家一定要重视上述讨论的问题，在程序中一定要注意动态内存管理问题。即使程序能在某些机器上正确地运行，但这并不能说明程序是正确的。在对程序进行测试时，一定

要注意这一点。

程序可能运行数月甚至数年也不出任何问题，但当安装了其他的应用程序或将系统升级到Windows™的更高版本时，改变了内存的使用，这时程序就崩溃了。或者它产生了错误结果，但是因为它成功运行了数月甚至数年而未曾被注意到。我们该怎么办？因为只是升级了操作系统便出错而诅咒微软？这不是微软的错，而是C++程序员的错，他忽略了在重载运算符函数operator+()的界面处放一个“&”符号。

下面的程序代码是该函数的正确形式，不是按值而是按引用来传递对象参数。

```
void String::operator += (const String &s)      // reference parameter
{ len = strlen(str) + strlen(s.str);            // total length
  char *p = new char[len + 1];                  // allocate enough heap memory
  if (p==NULL) exit(1);                         // test for success
  strcpy(p,str);                                // copy the first part of result
  strcat(p,s.str);                              // add the second part of result
  delete str;                                   // important step
  str = p; }                                    // now p can disappear
```

若程序11-3中的串接函数按引用来传递参数，则其输出结果如图11-11所示。

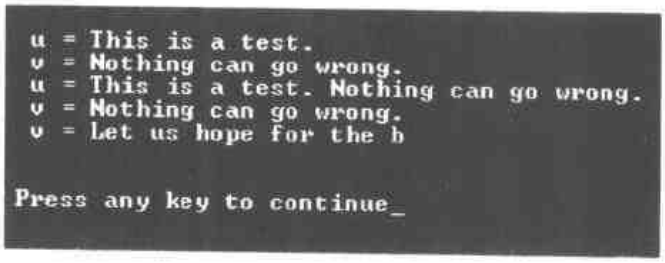


图11-11　其串接函数是按引用来传递参数的程序11-3的输出结果

不妨试着去运行该程序，以此实验来理解可能导致的问题。不要按值传递对象，除非别无选择。

在该源代码中增加或减少一个简单符号（&）竟然会如此大地影响程序的行为，确实让人惊心。注意，这两种不同版本的程序代码在语法上都是正确的，编译程序不会提示有问题。

按值传递对象参数仿佛驾驶着一辆坦克：可以到任何想去的地方，但也会引起许多间接的损失。正如我们在较早所提到的：不要按值传递对象，除非别无选择。

警告　不要按值向函数传递对象。如果对象有内部指针指向动态分配的堆内存，丝毫不要考虑把对象按值传递给函数，要按引用传递。并记住：若函数不能改变参数对象的状态和目标对象的状态，则要使用const修饰符。

11.3　对拷贝构造函数的深入讨论

让我们回顾一下，上一节所讨论问题的核心是拷贝其数据成员是指向堆内存的指针的对象。

假设每个对象实例都指向它所分配的内存区域，例如，String类有一指针指向堆内存区域，该区域中包含了与每个String对象有关的字符。

当一个对象的数据成员被拷贝到另一个对象数据成员时，这两个对象相应的指针将有相

同的内容。因此它们指向堆内存的同一区域。这两个对象在不同时刻撤销，例如，程序11-3中函数的形式值参数将在函数终止时消失，而实际参数仍然存在于客户空间，即main()函数中。当一个对象撤销时，其析构函数将删除由该对象指针所指向的内存。但第二个对象仍存在，而它毫无警告地丢失了它的堆数据。因此，对这个依赖于堆数据对象的任何使用都是不正确的，而且是“一个错误”。

如果还回给堆的这个内存没有马上为其他用途所用，该“幻觉”对象参与操作时，其被删除的内存就像依然存在一样。因此，测试的结果可能会使你认为程序是正确的。

当撤销第二个对象时，调用它的析构函数。注意在这里，我们并未说析构函数“再次被调用”。早些时候，另一个对象（形式参数）调用过析构函数，该对象已被撤销了。现在第二个对象（实际参数）调用析构函数，试图删除同一段堆内存。在C++中，这将导致错误，程序的行为未定义。也就是说，程序会失去控制。

11.3.1 完整性问题的补救措施

当动态分配内存的对象按值参数传递时，为避免可能出现的问题，有几种补救措施。

其中一种措施是去掉将堆内存还回给系统的析构函数。这既不是好的解决办法，也不是一个长久的解决办法。但这种方法可作为权宜之计，如果程序崩溃并需要运行程序以便调试，去掉析构函数可以让程序运行完。

另一种措施是在对象内使用固定大小的数组而不是使用动态分配的内存。这也不是巧妙的方法。但如果数组的大小足够，该方法是可行的。特别是当程序只处理比较少量的对象时，而且从程序完整性角度来看，偶然剪截长度超过固定大小的数据是可以接受时，该措施也是可行的。

对于参数传递，最好的方法是按引用来传递对象参数而不是按值传递。这样在拷贝对象时就不会有问题了。按引用来传递对象参数时，不再需要调用构造函数和析构函数来创建和撤销临时对象，因而可以加快程序执行速度。

不幸的是，该方法不是万能的。将一个对象拷贝到另一个对象而与参数传递无关时，就不能使用该方法。某类的一个对象由该类的其他对象初始化时就属于这种情况。下面这段程序代码，就是按引用方式将参数传递给函数operator+=()。

```
String u("This is a test. "), v("Nothing can go wrong.");
cout << " u = " << u.show() << endl;          // result is ok
cout << " v = " << v.show() << endl;          // result is ok
u += v;                                       // u.operator+=(v); by reference
cout << " u = " << u.show() << endl;          // result is ok
cout << " v = " << v.show() << endl;          // ok: pass by reference
v.modify("Let us hope for the best.");        // no memory corruption
String t = v;                                 // object initialization
cout << " t = " << t.show() << endl;          // ok: correct result
t.modify("Nothing can go wrong.");            // change both t and v
cout << " t = " << t.show() << endl;          // ok: correct result
cout << " v = " << v.show() << endl;          // v also changed
```

这段代码创建了两个String对象：u和v，以转换构造函数来对它们进行初始化，并将这两个对象连接起来。由于对象参数v是按引用传递给operator+-()，因而没有出现内存讹用现象，并且对象v保留其堆内存。当我们修改对象v时，只有对象v改变了，对象u没有变。

接着，我们创建了另一个String对象t，将它设置为v的当前状态。当我们修改对象t的内容时，希望对象v的值不变。图11-12给出了这段代码的预期执行结果。

```
u = This is a test.
v = Nothing can go wrong.
u = This is a test. Nothing can go wrong.
v = Nothing can go wrong.
t = Let us hope for the b
t = Nothing can go wrong.
v = Nothing can go wrong.
```

图11-12　上述客户代码段的预期（非实际的）输出结果

但实际上，事情并非总是如我们所期望的那样。String类（其参数按引用传递给重载运算符函数operator+=()）以及上面程序段的实现见程序11-4。我们对上面的程序段进行了修改，在嵌套作用域中创建对象t。当这段嵌套的程序终止时，对象t就会消失。我们可以通过检查对象v的状态来确认其完整性。程序11-4的真正执行结果如图11-13所示。

程序11-4　用一个对象的数据来初始化另一对象

```
#include <iostream>
using namespace std;

class String {
  char *str;                                  // dynamically allocated char array
  int len;
public:
  String (int length=0);                      // conversion/default constructor
  String(const char*);                        // conversion constructor
  ~String ();                                 // deallocate dynamic memory
  void operator += (const String&);           // concatenate another object
  void modify(const char*);                   // change the array contents
  const char* show() const;                   // return a pointer to the array
  } ;

String::String(int length)
{ len = length;
  str = new char[len+1];
  if (str==NULL) exit(1);
  str[0] = 0; }                               // empty String of zero length is ok, too

String::String(const char* s)
{ len = strlen(s);                            // measure the length of incoming text
  str = new char[len+1];                      // allocate enough heap space
  if (str==NULL) exit(1);                     // test for success
  strcpy(str,s); }                            // copy incoming text into heap memory

String::~String()
{ delete str; }                               // return heap memory (not the pointer!)

void String::operator += (const String& s)    // reference parameter
{ len = strlen(str) + strlen(s.str);          // total length
  char* p = new char[len + 1];                // allocate enough heap memory
  if (p==NULL) exit(1);                       // test for success
  strcpy(p,str);                              // copy the first part of result
```

```
  strcat(p,s.str);                          // add the second part of result
  delete str;                               // important step
  str = p; }                                // now temp can disappear

const char* String::show() const            // protect data from changes
{ return str; }

void String::modify(const char a[])         // no memory management here
{ strncpy(str,a,len-1);                       // protect from overflow
  str[len-1] = 0; }                           // terminate String properly

int main()
{ cout << endl << endl;
  String u("This is a test. ");
  String v("Nothing can go wrong.");
  cout << " u = " << u.show() << endl;      // result is ok
  cout << " v = " << v.show() << endl;      // result is ok
  u += v;                                   // u.operator+=(s);
  cout << " u = " << u.show() << endl;      // result is ok
  cout << " v = " << v.show() << endl;      // ok: pass by reference
  v.modify("Let us hope for the best.");    // no memory corruption
  { String t = v;                           // initialization
    cout << " t = " << t.show() << endl;    // ok: correct result
    t.modify("Nothing can go wrong.");      // change both t and v
    cout << " t = " << t.show() << endl;    // ok: correct result
   cout << " v = " << v.show() << endl; }   // v also changed
  cout << " v = " << v.show() << endl;      // t died, v is robbed
  return 0;
  }
```

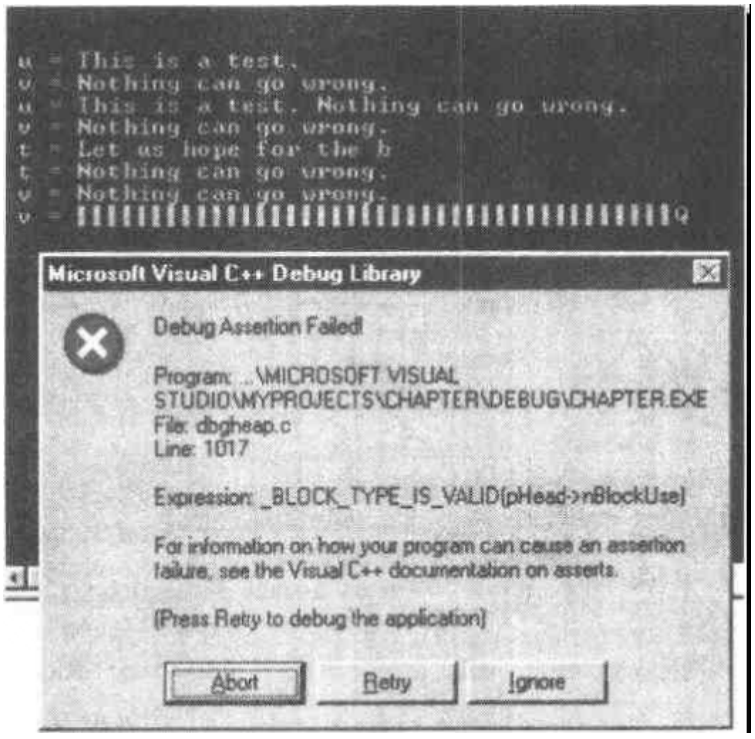


图11-13 程序11-4的输出结果

当创建String对象t时（因为t是一个在栈中创建的局部自动变量），分配了足够的内存

来存放字符指针和整数。接着，调用构造函数。在客户代码中将看到赋值符号，但它并不是赋值——而是初始化。正如我们较早提到的，是否在对象创建后调用构造函数并不是问题，问题在于调用哪一个构造函数。这个答案依赖于对象创建时客户代码所提供的数据。在程序11-4中，main()提供了一个实际参数，它是已存在的对象v。这样，构造函数带有一参数，这个参数是与构造函数所属的类型（在这里是String类）相同的对象。

如何称呼这样一个带有一个参数的构造函数呢？正如我们在第9章中提到的，它是一个拷贝构造函数，因为它将一个对象的数据拷贝给另一个对象。但String类并没有拷贝构造函数。尝试着使用这个String类不存在的拷贝构造函数会产生错误吗？不会。编译程序将生成一个对系统提供的拷贝构造函数的调用。编译程序提供该构造函数，并且编译程序生成该调用。该构造函数将参数对象域的内容拷贝给正在创建的对象。对于String类，系统提供的拷贝构造函数形式如下。

```
String::String(const String& s)  // system-provided constructor
{ len = s.len;                    // copy the length of the object text
  str = s.str; }                  // copy the pointer to the object text
```

该构造函数的工作过程如图11-14所示。创建String对象t时，设置它的len值为9，并设置str为与对象v的指针str所指向的同一个堆内存单元。

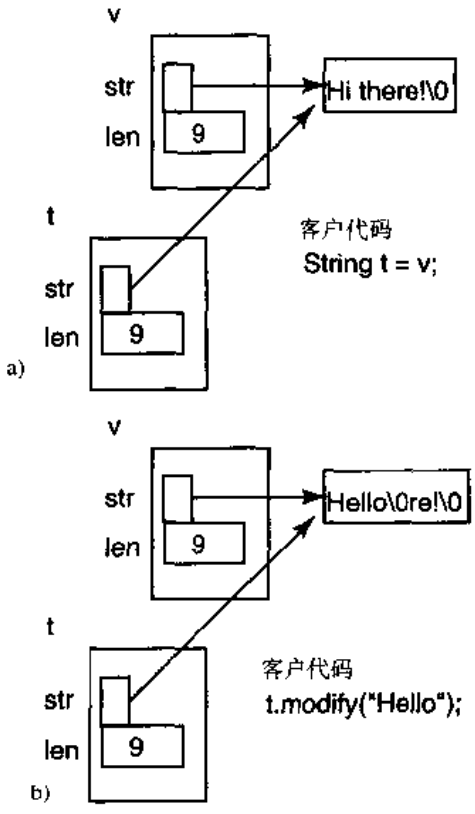


图11-14　用另一个String对象的数据来初始化一个String对象的内存图

类似于较早关于参数传递的论述，两个对象t和v共享同一段堆内存，这段堆内存事先分配给对象v，但现在对象t与之共享。而且每个对象都认为这段堆内存只属于自己。这样的情况比按值传递更糟。在按值传递中，实际参数存在于客户代码的作用域中，而形式参数存在

于服务器程序的作用域中。在执行的每一时刻，只有一个对象有效。而这里，两个对象都存在于相同客户代码的作用域中，且在相同的作用域中可被修改和访问。

由于这两个对象共享堆内存的相同区域，从客户代码的角度而言，它们是同义词。这正是客户代码修改对象t后，对象v也随着变化的缘故。图11-14能帮助我们更清楚地理解这一点吗？请看看图11-13的输出。按通常的编程直觉，对象v在客户代码中是没有理由改变的，但它还是改变了。

但只是按通常的编程直觉是不够的。在介绍编程的课程中，我们经常会碰到一些学生，连处理整数的简单代码都不能理解。

```
int v = 10; int t = v; t = 20;              // what is v now?
```

大多数程序员认为，显然在t改变后v没有变化，因为t和v在内存中的地址不同。另外一种观点：既然已经声明v和t这两个变量是一致的，那么改变了t后，v也应理所当然随之改变。

在某种意义上，他们的看法都有一定的道理。如果该变量是同义词，改变一个很显然也会改变另一个。回顾一下，一个是常规变量，而另一个是引用变量的情况是十分常见的。

```
int v = 10; int& t = v; t = 20;             // what is v now?
```

在这个例子中，不能按通常的编程直觉来考虑问题，而应该用初学者的思维逻辑。我们已经声明两个变量v和t是一样的，当然v会随着t的变化而变化。现在，v的值为20。这是所有C++程序员，无论是初学者还是编程专家，都应该接受的一个事实。

11.3.2 拷贝语义和值语义

实际上，有两种常见的编程直觉对应于计算机科学中的两个不同概念：*值语义*和*引用语义*（在这里，语义指的是拷贝数据的意思）。

比较常用的编程直觉是值语义。每个可计算的对象（如内部数据类型的变量或程序员定义类型的对象）在内存中有其自己独立的区域。使两个可计算对象相等意味着在另一个对象的内存中重复了相同的内容。在C++语言中（与在其他大多数程序语言中一样），内部数据类型的变量和程序员定义类的对象都使用值语义。

```
int v = 10; int t = v; t = 20;     // value semantics, v is 10
```

为什么值语义较常用呢？按其语义来说，两个对象的值相同，但它们的位模式（bit pattern）是独立分开的。改变一个对象的值并不会影响另一个对象中已存在的内容。

另一种较少用的编程直觉使用的是引用语义。当可计算对象被赋值时，它得到所赋值的引用（或指针）。使两个可计算对象相等意味着将它们的引用（或指针）都指向内存的相同单元。当一个对象所指向的字符数组改变了，另一个对象也会改变。这是因为它们的指针都指向相同的单元。在C++中，指针或引用要使用引用语义；按指针或引用传递参数时，或使用数组及带有指针链接的数据结构时也使用引用语义。

```
int v = 10; int& t = v; t = 20;    // reference semantics, v is 20
```

引用语义比较少用不足为奇。使用它主要出于性能的考虑（例如在传递参数时去掉对象的拷贝）。有时候，使用引用语义会出现非预期的效果，如上面这个例子。你必须随时意识到这一点并适当地进行处理。C++程序员应时刻注意值语义和引用语义的区别。

程序11-4中的问题还没有结束。当执行到嵌套作用域的闭花括号时，对象t将要消失，因为它是在该嵌套作用域中定义的。对象v是在包括main()函数的作用域内定义的，它仍可继续有效地使用。在程序11-4中，我们想在main()函数结束时打印v的值。注意这条语句与前面的打印语句是由嵌套域的闭花括号来分开的。表面上，似乎在客户代码的这两条语句之间并没有事件发生，因此它们的输出结果应是相同的——但实际却并非如此。这里，传统的编程直觉不足以理解一个C++程序。我们必须培养自己的直觉以帮助理解与之类似的代码段。

正如在图11-14中所看到的，第一条语句输出的结果是可识别的，虽然它不是我们所期望的，但是至少存在。第二条语句输出的却是不可识别的怪符号。那么这两条语句之间发生了什么事情呢？在嵌套作用域的闭花括号处，局部对象t调用String的析构函数，如程序11-4和图11-14所示，该析构函数删除对象t的指针str所指向的堆内存。该动态内存实际上是属于对象v的，但系统不能记住这点，它只知道应该按String析构函数的代码所示，删除指针str指向的内存。对象v被剥夺了动态内存，却不为人知。该对象形式上仍处于作用域内而且看起来十分正常。但这只是表面现象而已，客户代码不能再使用该对象了。

这种情况与按值传递参数类似。如按值传递对象一样，问题没有结束。当执行到闭花括号时，对象v应该按照作用域规则消失。消失之前调用析构函数试图删除已被删除的堆内存。这样程序出错，将会因失去控制而崩溃。

11.3.3 程序员定义的拷贝构造函数

除非放弃动态内存的管理问题，否则只有一个解决办法——使用程序员定义的拷贝构造函数。该构造函数应采用与上一节中所讨论的串接操作类似的方法，为目标对象分配堆内存。下面是其算法：

1) 将作为参数的字符数组的长度拷贝给目标对象的len。

2) 分配堆内存；并将目标对象的指针str指向所分配的堆内存。

3) 测试内存分配成功与否；如果系统内存不足则放弃操作。

4) 将目标对象的字符拷贝到新分配的内存空间中。

下面是程序员定义的用来解决上述问题的拷贝构造函数：

```
String::String(const String& s)       // programmer-defined copy constructor
{ len = s.len;                        // length of the source text
  str = new char[len+1];              // request separate heap memory
  if (str == NULL) exit(1);           // test for success
  strcpy(str,s.str); }                // copy the source text
```

注意这里的参数s是按引用来传递的，它是实际参数对象的引用。在传递参数时，参数的数据成员没有拷贝。相反，在构造函数内为目标对象分配了动态的内存。实际参数对象的动态内存拷贝给目标对象的堆内存。

这样做比程序11-4中按数据成员顺序进行的拷贝效率要低。值语义比引用语义速度要慢，因为它操作的对象是值，而不是引用或指针。但值语义比较安全。回顾一下导致这所有问题的客户代码。

```
String t = v;          // no problem if copy constructor is used
```

该代码执行后，两个对象t和v的指针str，将指向堆内存的不同区作用域。这样完整性

问题就解决了。

注意 如果类的数据成员中有指针，且该类的对象处理动态的堆内存，该类的设计者应决定它是需要值语义还是需要引用语义。如果我们需要值语义，并且用另一个对象的值去初始化一个对象，那么我们必须确信该类有程序员定义的拷贝构造函数。

程序11-5由程序11-4改进而来，它们的不同在于类String定义了自己的构造函数来支持对象初始化的值语义，其程序实现见程序11-5。其输出结果如图11-15所示。正如我们看到的，完整性问题没有了。String对象t和v不再是同义词。当对象t改变时，对象v则保留其原值。当嵌套作用域终止时，对象t消失，对象v仍存在，且能由客户代码继续使用。要跟踪代码及其输出以确认明白了对象t和v之间的关系。

程序11-5 使用拷贝构造函数用另一个对象的数据去初始化一个对象

```
#include <iostream>
using namespace std;

class String {
  char *str;                          // dynamically allocated char array
  int len;
  char* allocate(const char* s)       // private function
  { char *p = new char[len+1];         // allocate heap memory for object
    if (p==NULL) exit(1);             // test for success, quit if no luck
    strcpy(p,s);                      // copy text into heap memory
    return p; }                       // return pointer to heap memory
public:
  String (int length=0);              // conversion/default constructor
  String(const char*);                // conversion constructor
  String(const String& s);            // copy constructor
  ~String ();                         // deallocate dynamic memory
  void operator += (const String&);   // concatenate another object
  void modify(const char*);           // change the array contents
  const char* show() const;           // return a pointer to the array
  } ;

String::String(int length)
{ len = length;
  str = allocate(""); }               // copy empty String into heap memory

String::String(const char* s)
{ len = strlen(s);                    // measure the length of incoming text
  str = allocate(s); }                // allocate space, copy incoming text

String::String(const String& s)       // copy constructor
{ len = s.len;                        // measure length of the source text
  str = allocate(s.str); }            // allocate space, copy incoming text

String::~String()
{ delete str; }                       // return heap memory (not the pointer!)
void String::operator += (const String& s)     // reference parameter
{ len = strlen(str) + strlen(s.str);           // total length
  char* p = new char[len + 1];                 // allocate enough heap memory
  if (p==NULL) exit(1);                        // test for success
  strcpy(p,str);                               // copy the first part of result
  strcat(p,s.str);                             // add the second part of result
```

```
  delete str;                                    // important step
  str = p; }                                     // now pointer p can disappear

const char* String::show() const                 // protect data from changes
{ return str; }

void String::modify(const char a[])              // no memory management here
{ strncpy(str,a,len-1);                          // protect from overflow
  str[len-1] = 0; }                              // terminate String properly

int main()
{ cout << endl << endl;
  String u("This is a test. ");
  String v("Nothing can go wrong.");
  cout << " u = " << u.show() << endl;           // result is ok
  cout << " v = " << v.show() << endl;           // result is ok
  u += v;                                        // u.operator+=(v);
  cout << " u = " << u.show() << endl;           // result is ok
  cout << " v = " << v.show() << endl;           // ok: pass by reference
  v.modify("Let us hope for the best.");         // no memory corruption
  { String t = v;                                // call copy constructor
    cout << " t = " << t.show() << endl;         // ok: correct result
    t.modify("Nothing can go wrong.");           // change only t
    cout << " t = " << t.show() << endl;         // ok: correct result
    cout << " v = " << v.show() << endl; }       // v did not changed
  cout << " v = " << v.show() << endl;           // t died, v is intact
  return 0;
  }
```

```
u = This is a test.
v = Nothing can go wrong.
u = This is a test. Nothing can go wrong.
v = Nothing can go wrong.
t = Let us hope for the b
t = Nothing can go wrong.
v = Let us hope for the b
v = Let us hope for the b
```

图11-15　程序11-5的输出结果

在程序11-5中类String有三个构造函数，它们做的事情大致相同：分配堆内存并初始化其内容。在第一个转换构造函数中，初始化数据是一个空串（表示终止的0）；在第二个转换构造函数中，初始化数据是客户代码中用来作为实际参数的字符数组；在拷贝构造函数中，初始化数据是由客户代码提供的对象内部的一个字符数组。这个字符数组在堆中已被分配内存，它没有名字，由指向该数组的指针str来引用。由于参数对象s的类与正被初始化的目标对象的类String相同，拷贝构造函数有权使用其名字s.str访问该私有指针str。

不同的构造函数使用类似的算法是很自然的，因为创建对象时无论调用哪个构造函数，其结果对象都应该非常一致。当类有一个或两个构造函数时，可以一字不漏地重复使用该代码。当使用该公共的算法次数增多时（并且请放心，我们还没有完成），程序员常将该算法封装在一个私有函数中，并在不同的成员函数中调用它。该函数必须是私有的，因为客户代码并不关心对象内存的处理，这种低层的细节不应该困扰客户代码的算法和客户端代码程序员。我们可以在程序11-5中看到该私有函数。当它拷贝已分配的堆内存中的参数时，由于该数据

没有名字，它使用了指向堆内存的指针p的名字。因为这个数组还没有名字。

```
char* allocate(const char* s)  // private function
{ char *p = new char[len+1];   // allocate heap memory for object
  if (p==NULL) exit(1);        // test for success, quit if no luck
  strcpy(p,s);                 // copy text into heap memory
  return p; }                  // return pointer to heap memory
```

程序11-5中，第一个转换构造函数将一个空字符传递给allocate()函数；第二个转换构造函数将它自己的字符数组参数传递给allocate()函数；拷贝构造函数将其参数的字符数组s.str传递给allocate()函数。

当一个对象初始化另一个对象时，不可避免地要调用拷贝构造函数。问题是调用哪一个拷贝构造函数。如果该类不提供自定义的拷贝构造函数，编译程序将生成一个对系统提供的拷贝对象数据成员的拷贝构造函数的调用。如果该类的对象没有分配堆内存，这样是可行的。但如果对象使用了各自的堆内存段（值语义），使用系统提供的拷贝构造函数就会损坏应用程序的完整性。为了保持程序的完整性，类应该实现自己的拷贝构造函数，以便为目标对象提供自己的堆内存。

在以前的叙述中，“类应该实现”强调了在C++代码中不同程序段之间客户-服务器关系，也强调了人们关心的不同领域之间的客户-服务器关系。客户代码通过处理对象来表达自己的需求，以达到应用程序的目标（如用另一个对象初始化一个对象）；服务器代码通过实现客户代码调用的成员函数来支持客户代码的需求。隐式地调用构造函数，这并不改变客户-服务器关系。

当应用程序需要拷贝语义时，有动态内存管理的类可能被迫为一个对象初始化另一个对象的其他上下文提供拷贝构造函数。这样的上下文按值传递对象参数。如果有适当的拷贝构造函数，程序11-3中第一版的重载串接运算符函数operator+=()是相当不错的。

```
void String::operator += (const String s)    // pass by value
{ len = strlen(str) + strlen(s.str);         // total length
  char *p = new char[len + 1];               // allocate enough heap memory
  if (p==NULL) exit(1);                      // test for success
  strcpy(p,str);                             // copy the first part of result
  strcat(p,s.str);                           // add the second part of result
  delete str;                                // important step
  str = p; }                                 // now p can disappear
```

当调用该函数并创建了实际参数的备份时，将调用程序员定义的拷贝构造函数。它为形式参数s分配堆内存。该函数终止时为形式参数调用析构函数，删除它的堆内存，而不是实际参数的堆内存。这样，完整性问题就解决了。但性能问题尚未解决。按值传递参数时，调用串接运算符函数会涉及到创建对象、调用拷贝构造函数、分配堆内存、将一个对象的字符拷贝给另一个对象、调用析构函数以及回收堆内存。而按引用调用则不需要这些操作，引用语义去掉了不必要的拷贝所带来的额外开销。

警告 不要按值把对象传递给函数。如果对象有内部指针且动态处理堆内存，不要按值传递这些对象。但如果必须要按值来传递它们，则定义拷贝构造函数以解决完整性问题。要确保拷贝不会损害程序的性能。

11.3.4 按值返回

另一种使用值语义的情况是从函数中按值返回对象。我们已在第10章中对不处理动态内

存的类讨论了该问题。由于从函数返回对象与用另一个对象初始化一个对象的情况完全一样，我们在这里只作简单介绍。

程序11-6是String类的另一版本。我们在每一个构造函数中放入了调试跟踪语句，并加入了重载的比较运算符函数，将它作为一个成员函数来实现。此外，我们还添加了一个客户函数enterData()，并整理main()函数。该程序要求用户输入城市名，然后在数据库中查找该城市名。为简便起见，我们在main()函数中将数据库硬编码为一个字符数组的数组，并简单地使用顺序查找法在数据库中查找用户所输入的城市名。其执行结果如图11-16所示。

程序11-6　使用拷贝函数从函数中返回对象

```
#include <iostream>
using namespace std;

class String {
  char *str;                                  // dynamically allocated char array
  int len;
  char* allocate(const char* s)               // private function
  { char *p = new char[len+1];                // allocate heap memory for object
    if (p==NULL) exit(1);                     // test for success, quit if no luck
    strcpy(p,s);                              // copy text into heap memory
    return p; }                               // return pointer to heap memory
public:
  String (int length=0);                      // conversion/default constructor
  String(const char*);                        // conversion constructor
  String(const String& s);                    // copy constructor
 ~String ();                                  // deallocate dynamic memory
 void operator += (const String&);            // concatenate another object
 void modify(const char*);                    // change the array contents
 bool operator == (const String&) const;      // compare contents
 const char* show() const;                    // return a pointer to the array
 } ;

String::String(int length)
{ len = length;
  str = allocate("");                         // copy empty String into heap memory
  cout << " Originate: '" << str <<"'\n"; }

String::String(const char* s)
{ len = strlen(s);                            // measure length of incoming text
  str = allocate(s);                          // allocate space, copy text
  cout << " Created: '" << str <<"'\n"; }

String::String(const String& s)               // copy constructor
{ len = s.len;                                // measure length of the source text
  str = allocate(s.str);                      // allocate space, copy text
  cout << " Copied:  '" << str <<"'\n"; }

String::~String()
{ delete str; }                               // return heap memory (not the pointer!)

void String::operator += (const String& s)    // reference parameter
{ len = strlen(str) + strlen(s.str);          // total length
  char* p = new char[len + 1];                // allocate enough heap memory
  if (p==NULL) exit(1);                       // test for success
  strcpy(p,str);                              // copy the first part of result
```

```
  strcat(p,s.str);                          // add the second part of result
  delete str;                               // important step
  str = p; }                                // now pointer p can disappear

bool String::operator==(const String& s) const      // compare contents
{ return strcmp(str,s.str)==0; }            // strcmp returns 0 if the same

const char* String::show() const            // protect data from changes
{ return str; }

void String::modify(const char a[])         // no memory management here
{ strncpy(str,a,len-1);                     // protect from overflow
  str[len-1] = 0; }                         // terminate String properly

String enterData()
{ cout << " Enter city to find: ";          // prompt the user
  char data[200];                           // crude solution
  cin >> data;                              // accept user input
  return String(data); }                    // call the constructor
int main()
{ enum { MAX = 4} ;
  String data[4];                           // database of objects
  char *c[4] = { "Atlanta", "Boston", "Chicago", "Denver" };
  for (int j=0; j<MAX; j++)
  { data[j] += c[j]; }                      // data[j].operator+=(c[j]);
  String u = enterData();                   // crashes without copy con-
structor
  int i;
  for (i=0; i < MAX; i++)                   // i is defined outside of
the loop
    { if (data[i] == u) break; }            // break if String found
  if (i == MAX)                             // how did we get here?
    cout << " City " << u.show() << " is not found\n";
  else
    cout << " City " << u.show() << " is found\n";
  return 0;
  }
```

```
Originate:  ''
Originate:  ''
Originate:  ''
Originate:  ''
Created:  'Atlanta'
Created:  'Boston'
Created:  'Chicago'
Created:  'Denver'
Enter city to find: Boston
Created:  'Boston'
City Boston is found
```

图11-16 程序11-16的输出结果

在main()函数中创建对象数组时，数组中的每一个成分都调用缺省的String构造函数（例如带有缺省值的第一个转换构造函数）。该构造函数分配一长度为0的空字符串，并打印Originate消息。当调用operator+=()时，它将城市名追加到每个对象的内容中，该字符数组作为参数传递给比较运算符函数。该重载运算符函数期望一个String参数，这样，

每个数组成分调用第二个转换构造函数，并打印Created消息。

此后调用函数enterData()，它提示用户输入城市名，接收用户所输入的城市名，并将用户的输入作为一参数传递给String转换构造函数。我们可以看到这个构造函数打印出的Created消息。因为只有当调用enterData()函数时，才创建了main()中的对象u，enterData()中的构造函数调用相当于为main()中的对象u调用了构造函数。没有调用拷贝构造函数。尽管String对象处理动态内存，程序的完整性仍保持不变。这里的拷贝构造函数与值语义实现没有关系。转换构造函数的工作是将其单独的堆内存分配给main()中的对象u。正如第9章中的俄罗斯笑话：鳄鱼弹钢琴，鳄鱼唱歌，而猴子无事可做。

为了更方便动态处理管理内存的对象，我们在enterData()函数中作了一点小的修改——只不过添加了一个额外的局部对象来记录用户数据。

```
String enterData()
{ cout << " Enter city to find: ";          // prompt the user
  char data[200];                           // crude solution
  cin >> data;                              // accept user input
  String x = data;                          // conversion constructor
  return x; }                               // copy constructor
```

这个改动很小。如果x是一个内部数据类型的变量，那么这个改动近乎没有什么影响。但对于动态内存管理的对象而言，情形完全不同。在创建局部对象x时，转换构造函数被调用。然而当函数终止时，main()中的对象u由拷贝构造函数初始化。如果没有实现程序员定义的拷贝构造函数，则使用系统提供的拷贝构造函数。它将对象x的数据成员拷贝给对象u的数据成员，但不给u分配堆内存。对象u和x的指针str都指向同一堆内存。当enterData()函数终止时，对象x消亡，String析构函数被调用，它删除由对象x的指针str所指向的堆内存。这意味着对象u天生有缺陷，其动态内存在它创建时就删除了。

后果会如何呢？同前面的情形一样——机器崩溃了。有的机器也许会继续执行，但一切全是白费力气。程序不正确。它需要程序员定义的拷贝构造函数。

提供了程序员定义的拷贝构造函数后，一切就正常了。程序执行的示例结果如图11-17所示。

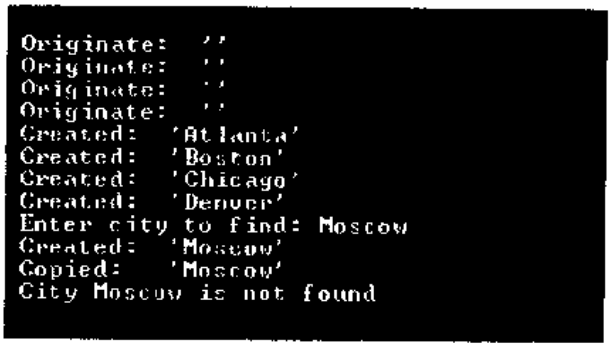


图11-17　修改了程序11-6的enterData()函数和加入了拷贝构造函数后的输出结果

调试输出结果表明，当用户输入信息行后，enterData()中的局部对象x调用转换构造函数。接着，为main()中的局部对象u调用拷贝构造函数。于是鳄鱼弹钢琴，猴子唱歌。这个版本的代码比上一版本的速度要稍慢一点儿，但这并不重要，重要的是它们的处理方式

不同，更为重要的是，如果x和u是内部数据类型的变量，这种改变并不会影响程序的执行。而且正是使用内部数据类型变量丰富了我们的编程经验。经过所有这些努力后，C++以不同的处理方式对待内部数据类型与程序员定义的类型。处理对象时需要改变编程直觉。这正是我们不辞辛苦、反复强调这些事件后果的原因：希望帮助大家培养这种新的编程直觉，以便能轻松地将客户代码结构与被隐式调用的类函数联系起来。

11.3.5 拷贝构造函数的有效局限性

现在我们几乎都清楚了，但还需要对程序做点修改。这次修改的是客户代码。在main()中，我们不再定义对象u和立即初始化u，而是先定义该对象（使用缺省的构造函数），在enterData()函数调用期间用户进行输入。

```
int main()
{ enum { MAX = 4} ;
  // setting up the database of city names
  // String u = enterData();        // crashes without copy constructor
  String u;                         // default constructor
  u = enterData();                  // It crashes! Copy constructor does not help
  // search for the city, printing the results
  return 0; }
```

这样修改后，我们的系统崩溃了。这次我们不再展示另一个显示问题起因信息的无用窗口。毕竟，这是在某台特定机器上，某个特定的操作系统下执行的情形。其关键问题在于程序本身是错误的。尽管在编译时正确，但它的行为没有定义，不应该运行。既然编译程序不会告诉我们程序是错误的，我们就得凭借编程经验来理解程序执行时潜在的问题。

11.4 赋值运算符的重载

我们曾多次提到，在C++中，对象的初始化与对象的赋值不同。处理内部数据类型时，两者之间的区别无关紧要。例如，考虑下面的客户代码段。

```
int v = 5; int u = v;              // variable u is initialized
```

将它与下面的程序代码作比较。

```
int v = 5; int u; u = v;           // variable u is assigned
```

在第一个例子中，变量u在定义时初始化。在第二个例子中，变量u在定义之后赋值。对于内部数据类型的变量，最后的结果是相同的。当这些可计算的对象是程序员定义类型的对象，且这些对象还处理各自的内存时，两者之间的区别就明显了。

```
String v = "Hello"; String u = v;       // object u is initialized
String v = "Hello"; String u; u = v;    // object u is assigned
```

在第一行代码中，如果类没有拷贝构造函数，我们就会陷入困境。在第二行代码中，如果类没有重载的赋值运算符，我们也会碰到麻烦。第二行代码将不会调用拷贝构造函数。

11.4.1 系统提供的赋值运算符的问题

如果类有重载的赋值运算符，上述客户代码例子中的第二行代码将调用该函数。如果类没有提供赋值运算符，编译程序会提供它自己的赋值运算符。该运算符与拷贝构造函数非常

类似：将赋值运算符右边对象的数据域拷贝给其左边对象的数据成员。

类似于系统提供的拷贝构造函数，系统所提供的赋值运算符通常是有效的。对于进行动态内存管理的类（如，Complex类、Rational类、Rectangle类），系统所提供的赋值运算符是够用的。但对于动态管理其内存的类而言，使用系统所提供的赋值运算符将会出现一些问题。

当在String对象上执行赋值运算时，数据成员是按顺序拷贝的。赋值运算符左边对象的指针str与其右边对象的指针str指向堆内存的相同区域。这两个对象成为同义词。如果修改了其中一个对象，例如u，那么另一个对象也随之改变，即此时v也改变了。

当其中一个对象，例如u，根据作用域规则或由delete操作撤销了，调用该对象的析构函数，删除对象指针str所指向的内存。结果，另一个对象，在此例中为v，尽管它看起来仍安然无恙地出现在程序中，但其堆内存已剥夺了。再使用该对象是不正确的。当该对象也撤销时，调用析构函数以试图删除由指针str所指向的堆内存。但该堆内存已删除了！正如我们在前面提到的，删除已被删除的堆内存将会导致不可预期的程序行为发生。该程序在语法上是正确的，但在语义上是错误的。

很难去跟踪问题产生的原因，因为问题的产生与程序执行的结果没有直接的联系。用拷贝构造函数也不能解决该问题，因为在执行赋值运算时没有激活构造函数。在C++中，赋值与初始化是不同的两个概念。

11.4.2　重载的赋值：内存泄漏

上述问题的解决办法是重载类的赋值运算符。重载的赋值运算符要确保运算符左边的对象与右边的对象最后不要指向相同的堆内存区域。

基本C++赋值运算符是带有两个操作数（运算符左边的操作数和右边的操作数）的二元运算符。程序员定义的赋值运算符也是这样的。于是，赋值运算符的界面与拷贝构造函数相似：左边的对象是消息的目标，右边的对象是参数。

```
u = v;                  // u.operator=(v);
```

这意味着类String的重载赋值运算符应具有如下形式的界面。

```
void String::operator = (const String& s);    // assignment operator
```

赋值运算符将参数对象的非指针数据成员拷贝给目标对象，并分配足够的存储空间，将参数对象堆内存中的内容拷贝给目标对象的堆内存。这些操作与拷贝构造函数的操作类似。

1) 将参数字符数组的长度拷贝给目标对象的len，
2) 分配堆内存；并将目标对象的指针str指向所分配的堆内存。
3) 测试内存分配成功与否；如果系统内存不足则放弃操作。
4) 将参数对象的字符拷贝给新分配的存储区域。

注意　如果需要将一个对象赋值给另一个对象，而且这些对象都动态管理堆内存，那么一定要有重载的赋值运算符。只有拷贝构造函数不足以应付问题。

下面是实现上述算法的赋值运算符。尽管它比系统所提供的赋值运算符的速度要慢，但它保持了值语义，使得两个对象相对独立。

```
void String::operator = (const String& s)
{ len = s.len;                          // copy non-pointer data
  str = new char[len + 1];              // allocate own heap space
  if (str == NULL) exit(1);             // test for success
  strcpy(str,s.str); }                  // copy heap data
```

这是一个很好的赋值运算符，但它在处理目标对象的方式上与拷贝构造函数完全相同，就好像目标对象没有任何以前的数据，是全新创建的一样。这种假设对于拷贝构造函数是成立的，但对于赋值运算符并非如此。目标对象u在早些时候就已创建了。这意味着在对象创建时调用了构造函数，在构造函数调用时，指针str已指向堆内存的某个单元。赋值运算符将忽略该堆内存，而将指针str指向堆内存的另一个单元。于是，早些时候分配给该对象的内存丢失了。在C++程序中，除了两次删除同一内存这个危险外，赋值运算符所带来的第二个危险是内存泄漏。

那么，有什么补救措施呢？与拷贝构造函数不同，赋值运算符必须将目标对象在赋值前所使用的资源（内存）释放掉。修正并不难，但必须知道一定要修正它。下面是修改了的更好一些的重载赋值运算符。

```
void String::operator = (const String& s)
{ delete str;                       // you do not do it in the copy constructor
  len = s.len;                      // copy non-pointer data
  str = new char[len + 1];          // allocate own heap space
  if (str == NULL) exit(1);         // test for success
  strcpy(str,s.str); }              // copy heap data
```

11.4.3 重载的赋值：自我赋值

在大多数情况下，前面所讲的赋值运算符已经够用了。但有些时候，可能会遇到一个不常见的问题：它并不支持客户代码以如下方式进行赋值：

```
u = u; // u.operator = (u); you do not do that often, do you?
```

这是无用的操作，但对于C++内部数据类型的变量则是合法的。因此没什么理由让它对于程序员定义类型的变量是非法的。事实上，它是合法的，编译程序并不会标注此语句有语法错误。只是operator=()函数的第一条语句删除了参数对象的堆内存，当执行库函数strcpy()时，它将新分配内存中的字符拷贝给它自己。在重叠的内存区域之间进行拷贝是没有定义的行为。（这又是一个令人头疼的问题）但即使定义了，对象堆内存中的字符已永远地丢失了。

这虽然有点儿奇怪，但并不意味着自我赋值不常用，在排序算法和指针操纵中经常用到它。为了防止对象堆内存的还回，赋值运算符可以测试参数对象的引用是否指向目标对象所处的同样地址。使用this指针访问目标对象的单元是个好办法。

```
void String::operator = (const String& s)
{ if (&s == this) return;           // avoid memory loss on self-assignment
  delete str;                       // you do not do it in the copy constructor
  len = s.len;                      // copy non-pointer data
  str = new char[len + 1];          // allocate own heap space
  if (str == NULL) exit(1);         // test for success
  strcpy(str,s.str); }              // copy heap data
```

当然，客户代码在调用赋值运算符前也可以进行该测试。但这样会将任务上托（pull up）

到客户代码，而不是将任务下推（push down）到服务器代码。

另一个办法是测试目标对象的指针str与参数对象的指针str是否指向同一堆内存区域。这样，在赋值运算符中测试语句如下所示：

```
if (str == s.str) return;          // same heap memory?
```

这两种方法是等价的，只是由于某种原因，第一种方法比较常用。其原因可能是对于C++程序员而言，this指针具有某些独特的美学价值。

11.4.4 重载的赋值：链表达式

上述赋值的运算符函数适用于动态处理其内存且需要支持赋值运算的所有类。但该赋值运算符并不支持链式表达式，即不支持在表达式中使用赋值运算的返回值。

```
t = u = v;          // returning void type does not support this
```

尚不清楚支持链式赋值运算到底有多重要。因为可以在客户代码中使用一系列的二元运算符来代替它。

```
u = v;              // binary operator: u.operator=(v);
t = u;              // binary operator: t.operator=(u);
```

在这里，又是一个同等对待内部数据类型的变量和程序员定义数据类型的变量的问题。对于内部数据类型的变量，链式赋值运算是有效的C++代码。那么它对于程序员定义数据类型的变量也应该是有效的。

赋值运算符是右结合的。赋值链的意义如下所示：

```
t = (u = v);        // t.operator = (u.operator = (v));
```

这意味着赋值运算符必须返回某个值，该返回值可以用作另一个赋值运算符（或另一个消息）的实际参数。因此，该返回值应属于赋值运算符所属的类型。

```
String String::operator = (const String& s)      // return an object
{ if (&s == this) return *this;          // protection against self-assignment
  delete str;                            // you do not do it in the copy constructor
  len = s.len;                           // copy non-pointer data
  str = new char[len + 1];               // allocate own heap space
  if (str == NULL) exit(1);              // test for success
  strcpy(str,s.str);                     // copy heap data
  return *this; }
```

程序11-7是程序11-6的修改版。我们加入了一个重载的赋值运算符。它调用私有函数allocate()以请求堆内存空间，并测试内存分配成功与否。为了减少调试输出的工作量，我们从缺省的构造函数中去掉了Originate消息。反而添加了Assigned消息，赋值运算符每次被激活时都会显示这个消息。而且，在录入城市名的数据库的循环中，我们不再调用串接运算符函数operator+=()，而是调用赋值运算符。该程序的输出结果如图11-18所示。

程序11-7　带有重载赋值运算符的String类

```
#include <iostream>
using namespace std;

class String {
  char *str;                                  // dynamically allocated char array
```

```
  int len;
  char* allocate(const char* s)          // private function
  { char *p = new char[len+1];            // allocate heap memory for object
    if (p==NULL) exit(1);                // test for success, quit if no luck
    strcpy(p,s);                         // copy text into heap memory
      return p; }                        // return pointer to heap memory
public:
  String (int length=0);                 // conversion/default constructor
  String(const char*);                   // conversion constructor
  String(const String& s);               // copy constructor
  ~String ();                            // deallocate dynamic memory
   void operator += (const String&);     // concatenate another object
   String operator = (const String&);    // assignment operator
   void modify(const char*);             // change the array contents
   bool operator == (const String&) const;  // compare contents
   const char* show() const;             // return a pointer to array
   } ;

 String::String(int length)
 { len = length;
   str = allocate(""); }                 // copy empty String into heap memory

 String::String(const char* s)
 { len = strlen(s);                      // measure the length of incoming text
   str = allocate(s);                    // allocate space, copy incoming text
   cout << " Created: '" << str <<"'\n"; }

 String::String(const String& s)         // copy constructor
 { len = s.len;                          // measure length of the source text
   str = allocate(s.str);                // allocate space, copy incoming text
   cout << " Copied:  '" << str <<"'\n"; }

 String::~String()
 { delete str; }                         // return heap memory (not the pointer!)

 void String::operator += (const String& s)  // reference parameter
 { len = strlen(str) + strlen(s.str);    // total length
   char* p = new char[len + 1];          // allocate enough heap memory
   if (p==NULL) exit(1);                 // test for success
   strcpy(p,str);                        // copy the first part of result
   strcat(p,s.str);                      // add the second part of result
   delete str;                           // important step
   str = p; }                            // now p can disappear

 String String::operator = (const String& s)
 { if (&s == this) return *this;         // test for self-assignment
   delete str;                           // you do not do it in copy constructor
   len = s.len;                          // copy non-pointer data
   str = allocate(s.str);                // allocate space, copy incoming text
   cout << " Assigned: '" << str <<"'\n";  // for debugging only
   return *this; }                       // return the target object to client

 bool String::operator==(const String& s) const      // compare contents
 { return strcmp(str,s.str)==0; }        // strcmp returns 0 if the same

 const char* String::show() const        // protect data from changes
 { return str; }
```

```
 void String::modify(const char a[])          // no memory management here
 { strncpy(str,a,len-1);                      // protect from overflow
   str[len-1] = 0; }                          // terminate String properly
String enterData()
{ cout << " Enter city to find: ";            // prompt the user
  char data[200];                             // crude solution
  cin >> data;                                // accept user input
  return String(data); }                      // conversion constructor

int main()
{ cout << endl << endl;
  enum { MAX = 4} ;
  String data[4];                             // database of objects
  char *c[4] = { "Atlanta", "Boston", "Chicago", "Denver" };
  for (int j=0; j<MAX; j++)
  { data[j] = c[j]; }                         // assignment:
data[j].operator=(c[j]);
  String u; int i;
  u = enterData();                            // it needs assignment,
                                                 no copy constructor
  for (i=0; i<MAX; i++)
  { if (data[i] == u) break; }                // if
(data[i].operator==(u))
  if (i == MAX)
    cout << " City " << u.show() << " is not found\n";
  else
    cout << " City " << u.show() << " is found\n";
  return 0;
  }
```

```
Created:  'Atlanta'
Assigned:  'Atlanta'
Copied:   'Atlanta'
Created:  'Boston'
Assigned:  'Boston'
Copied:   'Boston'
Created:  'Chicago'
Assigned:  'Chicago'
Copied:   'Chicago'
Created:  'Denver'
Assigned:  'Denver'
Copied:   'Denver'
Enter city to find: Denver
Created:  'Denver'
Assigned:  'Denver'
Copied:   'Denver'
City Denver is found
```

图11-18　程序11-7的输出结果

现在不再出现完整性问题。我们可以像处理基本数值类型的对象一样来处理String对象。可以创建这些对象而不对它们进行初始化；也可以用一字符数组来对它们进行初始化；也可以用另一个先前创建的String对象来对它们进行初始化。还可以用一个String对象来对另一个String对象进行赋值，就像它们是数字一样。注意在C++中并不允许以上述方式来处理数组，C++的数组是按引用语义而不是值语义来实现的。

如果觉得合适，还可以在该类中加入任意多的算术运算符（比如让String对象相加、减、

乘等）。当然，应该同时替维护人员考虑：不可以让代码不必要地难于理解。

总之，C++所提供的运算符重载功能为计算机程序设计的优雅性作出了重大的贡献。

11.4.5 程序性能的考虑

要想得到灵活性是要付出代价的。如果希望用另一个对象初始化一个对象（在定义对象时，或按值传递参数，或从函数中返回值），则必须提供一个拷贝构造函数。如果想用一个对象来对另一个对象进行赋值，则必须提供一个重载的赋值运算符。

由于在进行动态内存管理时常出现完整性问题，因此许多程序员为每个动态管理内存的类都编写了拷贝构造函数和赋值运算符。有时候他们甚至为那些不进行动态内存管理的类也这样做。实现这些函数并不需要太多精力，因此往往任由这种情况存在。

我们认为这是在回避问题。如果不在程序中加入大量无用的函数，开发人员就需要仔细研究客户代码的需求，并需要考虑不同设计方案所带来的不同后果。

为一个类提供过多的实际并不需要的成员函数也会带来一些问题。其中一个问题就是膨胀的设计。这是一个不能小看的问题。当维护人员（或客户端代码程序员）浏览程序时，这些没有用处的函数会分散他们的注意力，使之忽略更重要的细节。

另一个是性能问题。如图11-18所示，性能问题是很明显的。对于循环中输入字符串的每次赋值，除了赋值运算符外，还要调用两个函数调用：

1) 为operator=()的参数调用转换构造函数。

2) 调用operator=()函数本身。

3) 为赋值运算符的返回值调用拷贝构造函数。

尽管做了这么多的努力，类对象与内部数据类型的值之间仍然有很大的差别。在上述循环中，如果数组data[]和c[]中的组件是内部数据类型，则只是一条语句。但对于String类的设计来说，情况完全不同，这个循环体表示的是三个函数调用。

```
for (int j=0; j<MAX; j++)
{ data[j]=c[j]; }          // assign: data[j].operator=(String(c[j]));
```

值得注意的是，这些操作的开销都比较大。除了函数调用本身的开销外，每个操作都还需要分配堆内存空间，将参数字符串拷贝到堆内存，之后在析构函数调用时，再将堆内存还回给系统。对于那些支持值语义以使其两个操作数拥有各自的堆内存空间的赋值运算符，将这些操作都执行一遍是不可避免的。但还要为赋值运算符的参数和返回值再执行两遍这样的操作，就过多了。雪上加霜的是，客户代码没有使用由拷贝构造函数所产生的对象（因为引入返回对象的做法只是为了支持链式赋值），这个对象最终在析构函数调用后悄悄地删除了。

11.4.6 第一种补救措施：更多的重载

有两种办法可以提高重载的赋值运算符的性能。将赋值运算符的参数由String类型改为字符数组类型，可以去掉转换构造函数的调用。

```
String String::operator=(const char s[])   // array as parameter
{ delete str;                               // you do not do it in the copy constructor
  len = strlen(s);
  str = allocate(s);                        // allocate space, copy incoming text
  cout << "Assigned:  '" << str <<"'\n";    // for debugging
  return *this; }
```

如果赋值运算既要支持字符数组，又要支持String对象，就必须重载赋值运算符两次，分别以String对象和字符数组作为参数类型。加入了第二个赋值运算符的程序11-7的输出结果如图11-19所示。在第二个赋值运算符的调试消息中，我们加了几个空格来区别第一个赋值运算符（参数为String对象）所打印的消息与第二个赋值运算符（参数为字符数组类型）所打印的消息。

```
Assigned:       'Atlanta'
Copied:    'Atlanta'
Assigned:       'Boston'
Copied:    'Boston'
Assigned:       'Chicago'
Copied:    'Chicago'
Assigned:       'Denver'
Copied:    'Denver'
Enter city to find: Atlanta
Created:   'Atlanta'
Assigned:  'Atlanta'
Copied:    'Atlanta'
City Atlanta is found
```

图11-19　加入了第二个赋值运算符的程序11-7的输出结果

11.4.7　第二种补救措施：按引用返回

第二种改善性能的办法是去掉冗余的拷贝构造函数的调用。这样必须将按值返回替换为按引用返回。下面的例子中的赋值运算符以字符数组为参数，并按引用返回。

```
String& String::operator = (const char s[])  // return reference
{ delete str;                                 // you do not do it in the copy constructor
  len = strlen(s);
  str = allocate(s);                          // allocate space, copy incoming text
  cout << " Assigned: '" << str <<"'\n";      // for debugging
  return *this; }
```

对于参数为String类型的第一个赋值运算符也可以同样处理。当从函数返回引用时（详见第9章），必须确保在函数终止后，该引用仍指向一个有效的对象。在本例中这样做是安全的。所返回的引用是客户代码中赋值运算符左边的对象的引用。例如，前面循环例子中的data[i]。它在赋值运算符终止后仍存在，因为它是在客户代码作用域中定义的。注意对服务器程序作用域中所定义对象的返回引用，该对象在函数调用后就会消失。许多编译程序对此只给出警告消息或放任自流。

程序11-7中两个赋值运算符都返回对象引用的输出结果如图11-20所示。

```
Assigned:       'Atlanta'
Assigned:       'Boston'
Assigned:       'Chicago'
Assigned:       'Denver'
Enter city to find: Denver
Created:   'Denver'
Assigned:  'Denver'
City Denver is found
```

图11-20　程序11-7的输出结果，该程序添加了第二个赋值运算符、并按引用从赋值运算符返回String对象

我们似乎已将赋值运算符彻底地研究透了，其实不然。有些完美主义者认为这样仍不够，因为还无法防止客户端代码程序员做某些不必要的事情，如在所返回的String对象消失之前改变其内容等。例如，在C++中，下面的程序段对于程序11-7中的赋值运算符是合法的。

```
for (int j=0; j<MAX; j++)
  { (data[j] = c[j]).modify("A city nobody heard of"); }    // legal
```

这段程序将一个对象赋值给另一个对象，返回目标对象的引用，并立即发送修改消息，所赋的值将永远不会使用。这样做没有什么意义，因此应该标注为语法错误。为了产生语法错误，应该设置所返回的引用为常量引用。

```
const String& String::operator = (const char s[])    // too much?
{ delete str;                                  // you do not do it in the copy constructor
  len = strlen(s);
  str = allocate(s);                           // allocate space, copy incoming text
  cout << "Assigned:  '" << str <<"'\n";       // for debugging
  return *this; }
```

不一定非要这样处理不可，但是完美主义者坚持这样做。如果有些事情没有意义，则不应该让这些无意义的事情合法化。

11.5 实用性的考虑：实现什么函数

在处理动态内存管理时一定要加倍小心，每一步偏颇都有可能导致程序性能的降低或程序完整性的破坏。

许多程序员认为，每次设计动态管理内存的类时，该类必须完整地实现以下辅助成员函数：

- 缺省的构造函数。
- 多个转换构造函数。
- 拷贝构造函数。
- 多个重载的赋值运算符。
- 析构函数。

我们并不认为需要完全按照以上建议。这得根据客户代码的需求而定，也许只需要其中一部分函数就足够。提供错误的运算符界面不会导致完整性问题，但会影响程序的性能。重要的是应该理解本章所讨论的问题。这样才有助于根据实际任务（客户需求）选择成员函数，从而设计出既正确又有效率的类。当为程序自动提供全部的函数功能时，客户代码执行得很好，但可能会失去优势，忘记初始化与赋值之间的区别。这样是很危险的。

一定要确保为设计的类使用了正确的工具。出现问题时，要分析情况，使用调试语句，并画图，但不要用一些不必要的成分去扩充类。一定要根据所要解决的问题选择恰当的工具。一定要记住拷贝构造函数和赋值运算符所解决的问题不同，不能互相替换使用。

客户代码常常不需要用另一个对象去初始化一个对象，或用一个对象为另一个对象赋值。现在，我们假设要实现一个表示窗口的类。为简便起见，我们只考虑一个显示在窗口中的文本数据成员。这个Window类与String类非常相似。它包含了动态分配的字符数组、析构函数、串接运算符函数，串接运算符函数接收在窗口中将显示的字符数组，并将它添加到窗口的内容中去。

```
class Window {
  char *str;                                    // dynamically allocated char array
  int len;
public:
Window()
{ len = 0; str = new char; str[0]= 0 ; }        // empty String
~Window()
{ delete str; }                                 // return heap memory
void operator += (const char s[])               // array parameter
{ len = strlen(str) + strlen(s);
  char* p = new char[len + 1];                  // allocate enough heap memory
  if (p==NULL) exit(1);
  strcpy(p,str); strcat(p,s);                   // form data from components
  delete str; str = p; }                        // hook up str to new data
const char* show() const
{ return str; } } ;                             // pointer to contents
```

实现一个功能完备的窗口需要更多的数据成员和成员函数。但这个设计对于演示问题而言已经足够了。

当然，在应用程序中Window类的对象比String类的对象要少。而且，当创建Window对象时，其内容初始化为空，数据是在执行时添加的。

正如我们在本章开始处所提到的，这种类型的类对象不能按值传递。如果客户代码按值传递Window参数或不小心漏写了&符号而无意中按值传递了参数，该如何处理呢？

```
void display(const Window window)               // do not do that!
{ cout << window.show(); }
```

用另一个窗口去初始化一个窗口，或者用一个窗口给另一个窗口赋值显然毫无意义。

```
Window w1; w1 += "Welcome, Dear Customer!";     // reasonable usage
Window w2 = w1;                                 // unreasonable usage
w2 = w1;                                        // even less reasonable usage
display(w2);                                    // pass by value: slow
```

该程序段的第二行和第三行语句毫无意义。大多数程序员是不会这样写程序的。而且display()函数按值传递其参数。大多数程序员（尤其是读过本书的程序员）是不会这样写的。既然大多数程序员不会这样编程，是否意味着我们可以设计一个没有拷贝构造函数和赋值运算符的Window类呢？如果有人（没有读过本书的人）写出了类似的代码段，将会带来完整性问题并影响程序的性能。但在C++中，这些代码是合法的。

我们需要在Window类中写上一串长长的注释："亲爱的用户，请不要用另一个Window对象初始化一个Window对象，也不要用一个Window对象给另一个Window对象赋值。请不要按值传递一个Window对象给函数或从函数中按值返回Window对象。否则，程序会有问题。"与其这样，不如为保护客户代码做些其他工作。

其中一个办法是在Window类中添加拷贝构造函数和赋值运算符。这样即使程序员写的代码很糟糕，也至少不会出现完整性问题。

另一个办法是让糟糕的代码出现语法上的错误。这是一个很有趣的思路。在设计类时，将那些客户代码中不正确使用该类的对象的情况定义为语法错误。这就需要类的设计者决定哪些用法是不正确的。利用这种办法可以去掉那段长长的注释。

然而这不是件简单的事情。可以不用程序员定义的拷贝构造函数和赋值运算符，但是系

统会提供它自己的拷贝构造函数和赋值运算符，而正是这些系统提供的成员函数会给动态内存管理的类带来完整性问题。为了防止该问题，需要在类中加上程序员定义的拷贝构造函数和重载的赋值运算符，而且要使这些函数不能被客户代码使用。若客户程序调用这些函数，则会出现语法错误。

明白其中的意思吗？下面，我们一起来编写一个客户代码不能调用的函数。如何编写呢？一种可能的办法是将该函数定义为非公共的，让它成为一个私有（或保护）函数。

该方法的实现见程序11-8。拷贝构造函数和赋值运算符被定义为私有的。甚至没有必要去实现这两个函数。只要给出函数的原型，客户代码调用该函数将会出现链接错误。因为链接程序看不到函数的代码。编译程序将认为main()中的后三行语句是错误的。将拷贝构造函数和赋值运算符的声明注释掉，编译程序才能接收这三条语句，将实际上不合理的代码认为是合法的。

程序11-8 私有函数原型例子（将对对象的不正确处理视为非法的）

```
#include <iostream>
using namespace std;

class Window {
  char *str;                                  // dynamically allocated char array
  int len;
Window(const Window& w);                      // private copy constructor
Window& operator = (const Window &w);         // private assignment
public:
Window()
{ len = 0; str = new char; str[0]= 0 ; }      // empty String
~Window()
{ delete str; }                               // return heap memory
void operator += (const char s[])             // array parameter
{ len = strlen(str) + strlen(s);
  char* p = new char[len + 1];                // allocate enough heap memory
  if (p==NULL) exit(1);
  strcpy(p,str); strcat(p,s);                 // form data from components
  delete str; str = p; }                      // hook up str to new data
const char* show() const
{ return str; } } ;                           // pointer to data
void display(const Window window)            // do not pass objects by value
{ cout << window.show(); }

int main()
{ Window w1; w1 += "Welcome, Dear Customer!\n";     // reasonable
  Window w2 = w1;                            // unreasonable usage: syntax error
  w2 = w1;                                   // even less reasonable usage: syntax
error
  display(w2);                               // pass by value: syntax error
  return 0;
  }
```

这是一个防止客户代码滥用类的好方法。当然，如果因为某种原因需要支持程序11-8的main()中标注为不合理的代码，或者如果比较关注性能问题，就必须为类提供拷贝构造函数和赋值运算符。如果右值表达式的类型不同，还要提供多个相应的赋值运算符。就转换运算符函数而言，如果必须用简单的数据对象来初始化类的对象，而不是用同一类型的对象来

初始化，则要提供转换运算符函数。提供转换运算符函数的另一个合理原因是避免多次重载运算符函数。在类中减少了函数的使用，其结果是要调用额外的构造函数和进行额外的内存分配操作。

11.6 小结

本章中我们对C++的阴暗面进行了分析讨论。我们并不是为了恐吓，而是为了强调C++程序员的艰巨任务：一定要注意程序的性能和完整性问题。

我们详细讨论了按值传递参数所带来的不良后果。我们希望编写程序时不要再按值传递参数。要按引用传递参数，并使用const修饰符标明参数在函数中不能修改。

我们不赞同按值从函数中返回对象。如果必须返回一个对象，则返回该对象的引用，并确保所引用的对象不在函数调用后就立即消失。

如果坚持不让客户代码按值传递类的对象，要将拷贝构造函数定义为私有函数。即在类的私有成员声明处加入拷贝构造函数的原型。没有必要去实现函数本身。

如果类动态地管理内存，则该类应该有析构函数来把空间还回给堆内存。

如果在客户代码中必须用类的一个对象去初始化该类的其他一个对象，则该类应有实现值语义的拷贝构造函数，为每个对象提供各自的堆内存空间段。

如果在客户代码中必须用类的一个对象给该类的另一个对象赋值，则必须提供实现值语义的重载赋值运算符，使每个对象拥有各自的堆内存空间段。在赋值运算符中，要确保在对象被参数对象赋以新值前还回原来占用的堆内存，以避免内存泄漏。在自我赋值时，要注意不要删除内存。还要考虑是否有必要去支持链式赋值。客户代码常常不需要链式赋值。

转换构造函数允许放宽C++中的强类型规则。使用转换构造函数，可以传递一个与类所要求类型不同的实际参数数据，而结果又是有效的。该方法虽好但要慎用。过多调用转换构造函数的开销很大，尤其是在必须使用值语义时。

而且，要分清楚客户代码在哪里调用了拷贝构造函数，在哪里调用了赋值运算符。它们都使用相同的符号“=”表示其操作，但却激活服务器代码中的不同函数。要注意两者的区别。

要反复阅读本章内容。画内存图来理解代码。要记住C++中的动态内存管理常常容易出现问题。在C++中，除了传统的语法错误和语义（运行）错误外，还有另外一种错误：程序在语法和语义上都是正确的，但程序本身却是错误的。没有其他语言要求程序员承担这么多的任务，一定要小心谨慎。

第三部分　使用聚集和继承的面向对象程序设计

这部分我们继续讨论面向对象的程序设计技术。C++为程序员增加了强大的技术支持，例如类的复合和类继承。但是，有些程序员在使用什么技术以及如何避免程序复杂性问题时，仍然会觉得难以抉择。

第12章讲述将对象作为另一个类的成员的语法，介绍怎样访问这些对象及其数据成员的一些规则，并解释怎样去初始化复杂对象的各个组件。本章还介绍了通过引用成员和静态成员去共享对象组件的技术，描述使用嵌套类和友元类的方法。

第13章介绍继承的使用技术。这一章描述了C++中继承的语法，讨论继承的不同方式和它们对在派生对象中访问基本成员的权限的影响。并且定义了在C++中继承所涉及到的作用域规则，并对派生方法隐藏了与它同名的基类方法时的名字解析规则作了介绍。本章还包括了派生对象的构造和析构规则以及执行构造函数与析构函数的顺序。

第14章介绍了统一建模语言（Unified Modeling Language，UML），它是一种越来越流行的描述面向对象设计的语言。这一章有助我们在使用继承和类的复合这两者中做出选择，并且介绍了根据类的能见度标准及类之间的任务划分来决定是使用继承还是使用类复合。这一章的重要性不仅在于它介绍了统一建模语言，而且也是因为它提醒我们不要过度使用继承，通常，过度使用继承将会导致程序复杂性的增加。

第12章　复合类的优缺点

本书的前两部分主要讨论了C++语言的规则，介绍了能做什么，不能做什么以及要注意避免哪些危险以免降低程序的性能或破坏程序的完整性。在这些部分中展示了C++作为一种功能强大的语言的特点，让程序员对C++程序及其执行情况有全面的了解。

本书第二部分介绍了与编译C++程序代码相关的面向对象程序设计基本原则，并分析了类与类之间的交互关系。内容包括：

- 在类中绑定数据与函数来表示这些数据与函数在逻辑上的所属关系。
- 将类的一些成员（数据和函数）定义为私有的，因为类外部访问这些成员将会使用户依赖于类设计的低层细节。
- 将类作用域作为消除不同类成员之间名字冲突的机制，让类的开发者决定如何去消除冲突。
- 提供成员函数使客户代码可以不必直接访问服务器的数据作用域名。
- 将客户的任务推到服务器的类及其成员函数中。
- 以调用服务器方法的方式来编写客户代码，这样可以使程序的开发不依赖于服务器的设计细节。
- 提供构造函数和析构函数，以实现合适的对象初始化、资源管理和将任务推到服务器

完成。

- 让程序维护者和客户知道服务器设计者的意图。例如，对数据成员、参数、返回值及方法使用const修饰符。

利用这些程序设计的基本思想可以提高面向对象的程序代码可读性及维护性。也只有利用这些程序设计的基本思想后才能认识到C++语言的无穷潜力。没有这些基本思想，程序代码之间将会极大地相互依赖，从而缠结不清。这样的代码无论是用C++、Java、COBOL还是FORTRAN编写，其可读性都会很差，难于修改。

在本部分，我们将不只考虑一个单独的C++类，而是讨论包含有相互协作的类的程序设计问题。我们介绍了类复合（class composition）的概念，即某个类的对象作为另一个类的数据成员、局部变量或方法的参数。类复合是组织程序类之间进行合作的强有力的技巧。有两个支持类复合的基础，一是C++语言的构造函数调用规则，另外一个是数据从客户代码传递给程序员定义类的组件的语法。

实现类协作的另一种方法是使用继承，每个类的设计方法都是如此相似的，以至于一个类只是比另一个类多了一些数据成员和方法。继承为C++中的代码重用提供了主要的支持。我们不仅讨论何时使用继承的设计问题，还将讨论所有支持继承的C++语言特征：继承语法、对象实例、传递数据初始化所继承的数据成员、名字二义性及解决二义性的规则。

C++程序员喜欢使用继承。许多专家认为继承的使用是面向对象程序设计的支柱。实际并非如此，使用C++类才是面向对象程序设计的中枢，它将数据和操作绑定在一起，并对类成员的访问实施控制等。

继承不是面向对象程序设计的支柱，它是代码重用和设计重用技术，因此对C++编程十分重要。我们必须正确地使用该技术。

12.1　用类对象作为数据成员

我们在第9章讨论了在C++中将类作为模块化的单位。C++构造类的主要目的是让程序员将逻辑上属于一个整体的数据和操作绑定在一起。

在本书前面有关C++类的大部分例子中，类的数据成员都是内部数据类型——整数与浮点数。一些比较复杂的例子中数据成员为字符数组，它实际上是指针，指向堆中所分配的字符数组。从构成类的角度来看，指针与整数、浮点数是类似的，没有从类之外可以进行访问的内部结构。

但不能因此将这一点作为设计类时的固有限制。类成员可以比内部数据类型的数据值更复杂。这也引入了一种学习语言的循序渐进的方法：先学习其简单特点，再进一步学习较复杂的特点。

在第10章与第11章中，我们可以看到，C++努力将内部数据类型和程序员定义的类型同样对待。如果内部数据类型的数据成员可以作为类的组件，那么一个类的数据成员没有理由不能是其他已拥有类的对象。

在C++中，可以用类的对象作为其他类的对象的组件。如果一个类有许多数据成员，可以将一些相关的数据成员组合成一个对象，并声明该对象为类的一个成员。这样可以用具有较少组件的类来代替具有许多组件的类。这样有利于在程序开发过程中程序员之间的分工协作，同时还有助于改善程序代码的模块化程序，有利于信息隐藏。过分进行模块化的坏处是代码

的用户会面对大量的小的类，从而导致学习这样的类库更加困难。

将其他类的对象作为其数据成员的类称为复合类（composite class）。虽然所有的类都有组件（数据成员），并由组件复合而成的。但这里的复合类主要指的是这些类中，其组件本身还有它自己的组成成分。在面向对象设计理论中，将一个类的对象作为其他类的对象的组件被称为类聚集（class aggregation）或类复合（class composition）。

下面的例子中，Rectangle类含有分别表示左上角和右下角的x、y坐标。这是图形程序设计中的常规处理。

```
class Rectangle {
  int x1, y1;                                  // coordinates of the top-left point
  int x2, y2;                                  // coordinates of the bottom-right point
  int thickness;                               // thickness of the rectangle border
public:
  Rectangle (int inX1, int inY1, int inX2, int inY2, int width=1);
  void move(int a, int b);                     // move rectangle
  void setThickness(int width = 1);            // change thickness
  bool pointIn(int x, int y) const;            // point in rectangle?
  . . . . } ;                                  // the rest of class Rectangle

Rectangle::Rectangle (int inX1, int inY1, int inX2, int inY2, int width)
{ x1 = inX1;  y1 = inY1;
  x2 = inX2;  y2 = inY2;
  thickness = width; }                         // set data members

void Rectangle::move(int a, int b)
{ x1 += a;  y1 += b;
  x2 += a;  y2 += b; }                         // move each corner

void Rectangle::setThickness(int width)
{ thickness = width; }                         // do the job

bool Rectangle::pointIn(int x, int y) const // is point in?
{ bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }
```

该类提供的服务有：沿着屏幕移动矩形对象、改变矩形边的宽度、检查所给定的点是否在矩形内等。在客户中可以指定矩形的角坐标来定义Rectangle类的对象。也可移动某点及矩形并试着去捕捉它们。

```
int x1=20,y1=40; int x2=70,y2=90;          // top-left/bottom-right corners
int x=100, y=120;                          // point to catch by the rectangle
Rectangle rec(x1,y1,x2,y2,4);              // create a Rectangle object
rec.setThickness();                        // line width is 1 pixel (default)
x -= 25;  y -= 15;                         // move the point around the screen
rec.move(10,20);                           // 10 pixels to right, 20 pixels down
if (rec.pointIn(x,y)) cout << "Point is in\n";        // in point in rectangle?
```

这个例子虽然很小，但让我们仍能感觉到Rectangle类的内部结构的复杂。在编写这些代码时，很容易将x1和y1，x1和y2等混淆。对于客户端代码程序员来说，定义Rectangle对象的工作太复杂了（在调用构造函数时有5个参数）。该Rectangle类之所以复杂，是因为它缺少一个组件：Point类。实际上，这里出现点的概念是很自然的，无论是在Rectangle

类中还是在客户中都出现了点的概念，但是程序员定义的类型中没有支持这个概念。

12.1.1 C++类复合的语法

我们继续讨论上述例子，并使用Point类来提供一些服务。与前面相同，我们假设下面的程序代码是一个很长的有许多人分工协作的程序的一部分。本节我们将集中讨论类复合的语法，以及类与类之间通信、设计这些类的人之间通信的有关问题。

```
class Point {
private:
  int x, y;                                  // private coordinates
public:
  Point (int a, int b)                       // general constructor
    { x = a;  y = b; }
  void set (int a, int b)                    // modifier function
    { x = a;  y = b; }
  void move (int a, int b)                   // modifier function
    { x += a;  y += b; }
  void get (int& a, int& b) const            // selector function
    { a = x;  b = y; }
  bool isOrigin () const                     // predicate function
    { return x == 0 &&  y == 0; } } ;
```

在这里我们使用一些常用的成员函数术语。修饰符（modifier）是改变目标对象状态的成员函数（确保没有const关键字）。选择符（selector）是不改变目标对象状态的成员函数（确保有const关键字）。谓词（predicate）是返回一个布尔值的选择符，该布尔值表示目标对象状态的有关信息（在本例中，表示它是否是原点）。

在这个例子中，使用成员函数类属名来表明类作用域，有效地限制了程序中的名字冲突。我们选择set()函数作为Point的一个成员函数时，并不需要通知所有为这个应用程序设计其他类的开发组成员。他们也可以在自己的类中使用名为set()的函数。我们只需要通知极少数开发组成员就可以了，因为他们设计的类会使用我们的类Point帮助实现其功能。其中一个这样的客户类是本章开始时所介绍的Rectangle类。这个Rectangle类有两个Point类数据成员，表示矩形的左上角与右下角坐标。数据成员thickness的意义与前面相同，表示在屏幕上画矩形的线的宽度。

```
class Rectangle {
  Point pt1, pt2;                            // top-left, bottom-right corner points
  int thickness;                             // thickness of the rectangle border
public:
  Rectangle (int inX1, int inY1, int inX2, int inY2, int wid=1);
  void move(int a, int b);                   // move both points
  void setThickness(int width = 1);          // change thickness
  bool pointIn(int x, int y) const;          // point in rectangle
  . . . . } ;                                // the rest of class Rectangle

Rectangle::Rectangle (int inX1, int inY1, int inX2, int inY2, int width)
  { pt1.set(inX1,inY1);  pt2.set(inX2,inY2);// push job down
    thickness = width; }                     // set data members

void Rectangle::move(int a, int b)
  { pt1.move(a,b);  pt2.move(a,b); }         // pass buck to members
```

```
void Rectangle::setThickness(int width)
  { thickness = width; }                          // do the job

bool Rectangle::pointIn(int x, int y) const  // is point in?
  { int x1,y1,x2,y2;                              // coordinates of corners
    pt1.get(x1,y1);  pt2.get(x2,y2);              // get point data
    bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
    bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
    return (xIsBetweenBorders && yIsBetweenBorders); }
```

可以看到，这个Rectangle类中有重要的改变。本来想使用“显著改变”的，但是考虑到对于这样一个小类来说不太合适。不管怎样，这些改变代表了类复合程序设计中的常规。

Rectangle类构造函数不再对内部数据类型的数据成员进行赋值，而是使用了传递给组件对象的两条消息。

```
pt1.set(inX1,inY1); pt2.set(inX2,inY2);    // push job down
```

该例将客户代码（Rectangle类）与服务器代码（Point类）的设计细节隔离开来。这里客户代码的实现基于发送给服务器对象的消息，客户并不关心服务器代码的设计细节，即客户代码只是关心正在做什么而不关心如何做。具体操作的细节从Rectangle类推到Point类。以这样风格编写的客户代码比较容易理解。

move()方法体现了C++中复合类和组件类之间关系的一个更有趣的惯用方法。当要移动一个Rectangle对象时，该对象要求其组件调用函数名也为move()的方法。但这并不是递归调用相同的函数。第二个move()方法是组件类Point的方法，并不属于复合类Rectangle。这个例子表明，在C++程序中将不同特点的对象做同样处理。在这里，同样处理指的是同名方法属于具有相似行为的类。移动一个矩形意味着移动矩形中的每个点。因此我们将两个方法都命名为move()，而不是movePoint()和moveRectangle()。

12.1.2 访问类数据成员的数据成员

上述Rectangle类的两个实现版本的另一个重要的区别是访问组件的组件。在第一个版本中，Rectangle类对x和y坐标可做任意处理，x和y是可直接访问的。在Rectangle的第二个版本中，x、y是Point类的组件。如果组件对象（在此例中指的是Point类）有公有的组件，复合类（Rectangle类）通过使用点选择运算符能访问其对象数据成员的数据成员（x和y）。也就是说，如果Point组件是公有的，Rectangle成员函数Rectangle::pointIn()可以使用Point组件已确认的名称。因此，Rectangle类能决定参数x是否在Rectangle数据成员pt1的x坐标和pt2的x坐标之间。

```
bool xIsBetweenBorders = (pt1.x<x && x<pt2.x)
            || (pt2.x<x && x<pt1.x);
```

但是，Point数据成员是私有的，且客户类（在此例中是Rectangle）没有访问服务器类（Point）组件的特权。Rectangle类将Point对象作为它自己的组件，因此，其方法可以访问其Point类数据成员（pt1和pt2）。但Rectangle类的方法不能访问Point组件的数据成员x和y。上面的这条语句是不合法的。Rectangle方法应使用Point公有成员函数来访问Point组件。例如，Point::get()。

重要的是不要混淆这两种情况。（已经多次强调这一点）。Rectangle类可任意访问其私

有成员Point类的pt1和pt2，但不能访问其数据成员的私有组件pt1.x、pt1.y、pt2.x和pt2.y。因此，Rectangle:: pointIn()必须按以下方式去获取Rectangle数据成员pt1和pt2的数据成员。

```
bool Rectangle::pointIn(int x, int y) const
{ int x1,y1,x2,y2;                          // coordinates of corners
  pt1.get(x1,y1);  pt2.get(x2,y2);          // get point data
  bool xIsBetwecnBorders = (x1<x && x<x2) || (x2<x && x<x1);
   ... }                                    // and so on
```

访问类数据成员的组件时需要使用服务器类的访问函数，因此，复合类方法的设计可能会相当令人讨厌。

警告 如果一个类的数据成员是其他的类，那么类成员函数不能访问这些数据成员的私有组件。这会带来很大的不便。但复合类必须使用数据成员的方法去访问它自己组件的组件。

下面再看一下Rectangle类的客户代码设计，它不需要任何改变。客户要给Rectangle构造函数提供5个参数，给Rectangle::pointIn()方法提供两个参数。这意味着组件类Point只是给复合类Rectangle的设计带来了方便，但对客户代码的设计没有任何影响。

```
int x1=20, y1=40; int x2=70, y2=90;      // rectangle  corners
int x=100, y=120;                        // point to catch with the rectangle
Rectangle rec(x1,y1,x2,y2,4);            // create a Rectangle object
rec.setThickness();                      // line width is 1 pixel (default)
x -= 25;  y -= 15;                       // move the point around the screen
rec.move(10,20);                         // 10 pixels to right, 20 pixels down
if (rec.pointIn(x,y)) cout << "Point is in\n";   // in point in rectangle?
```

在这个微小的例子中，区别并不明显，但已经很清楚了。与Rectangle类的第一个版本类似，该客户代码根据单独的实体x和y来进行处理。而不将这些单独实体聚集为一个类。因此也不需要将程序设计者的意图传达给维护者，无需让他们了解这些单独实体是相互关联的，表示的是同一个点的坐标。无论客户代码想如何处理点，如移动、比较等，其行为都是基于客户代码中某个坐标而进行的。较低级别的个体行为使客户代码混乱，难于理解。例如，客户代码语句rec.move(10,20);清楚地说明了矩形的移动。事实上，被移动的坐标为（100，120）的点必须由一系列的赋值x - = 25 ; y - = 15;来进行推理。较低级别的细节没有被推给服务器代码。

根据Point类对象及其操作来编写客户代码会减少这些问题，使得程序更加面向对象。

```
Point p1(20,40), p2(70,90);                  // rectangle  corners
Point point(100,120);                        // point to catch with the rectangle
Rectangle rec(p1,p2,4);                      // see below about problems with this
rec.setThickness();                          // line width is 1 pixel (default)
point.move(-25,-15);                         // move the point around the screen
rec.move(10,20);                             // 10 pixels to right, 20 pixels down
if (rec.pointIn(point)) cout << "Point is in\n";     // is point in?
```

这段程序代码有两个服务器类：类Point与类Rectangle。Point对象point的移动与Rectangle对象rec的移动一样清晰可见。较低级别的细节推给了服务器代码，客户代码中

的函数调用其义自明，不用再分别进行计算。

这段程序代码中的问题是所期望的Rectangle类的界面是无效的。Rectangle类所提供的构造函数的参数为5个，而客户代码只提供了3个参数；Rectangle类的函数pointIn()的参数为两个，而客户代码只提供了1个，解决该问题需要改变客户代码中的函数调用或者改变服务器类Rectangle中的函数接口。程序越小，所做的修改就越少。

如果Rectangle类是一个不能改变的库中的类，我们将没有选择。客户代码只能根据库的限制进行处理。如果Rectangle类是止为某个应用程序而开发的一些协作类中的一个类，该Rectangle类也可以改变，那么选择就是一个思想意识的问题了。从面向对象的思想意识方面来看，应该是服务器类（Rectangle类）去适应并满足客户代码的期望和需求，根据该观点，Rectangle类应按以下方式来实现。

```
class Rectangle {
  Point pt1, pt2;                                  // rectangle corners points
  int thickness;                                   // thickness of the rectangle border
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                         // move both points
  void setThickness(int width = 1);                // change thickness
  bool pointIn(const Point& pt) const;             // point in rectangle?
   . . . . } ;                                     // the rest of class Rectangle

  Rectangle::Rectangle (const Point& p1, const Point& p2, int width)
   { pt1 = p1;  pt2 = p2;
     thickness = width; }                          // set data members

  void Rectangle::move(int a, int b)
   { pt1.move(a,b);  pt2.move(a,b); }              // pass buck to Point

  void Rectangle::setThickness(int width)
   { thickness = width; }                          // do the job

bool Rectangle::pointIn(const Point& pt) const     // is point in?
{ int x,y,x1,y1,x2,y2;                             // coordinates of pt and corners
  pt.get(x,y);                                     // get parameter's coordinates
  pt1.get(x1,y1);  pt2.get(x2,y2);                 // get both corners
  bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }
```

注意Point类和Rectangle类中const关键字的使用，它反映了目标对象和函数调用参数的改变情况。由于这些类的成员函数不返问指针、引用或对象，因而没有必要使用const限制返回值。

Rectangle类的构造函数既可以只用两个参数来调用，也可以用3个参数来调用。当用两个参数调用时，thickness数据成员被设为缺省值1。

12.1.3 访问方法参数的数据成员

注意在C++中，方法参数的处理方式与复合类的数据成员的处理方式相似。

成员函数的参数可以是任意类型，包括类对象。类对象的参数使用模式没有任何限制：它们可以按值传递，也可以按指针或按引用传递，在必要时甚至可用const来修饰。

访问成员函数中参数对象的原则与访问其他对象一样：只允许访问公有成员。方法可以访问参数，但不能访问参数的私有组件。如果参数的客户（在这里指的是成员函数）需要访问服务器类的私有成员（参数），则必须使用服务器类的成员函数。

因此，Rectangle成员函数pointIn()使用Point访问函数get()去访问其参数的组件pt.x和pt.y。

在C++中，如果正被访问的另一个对象是相同类类型，则是一个重要的例外。将一个对象作为参数传递给该对象所属类的成员函数时就是这种例外情况。如果客户类与服务器是相同类型，那么客户对象拥有访问参数对象的组成成分的所有权限。上述例子中的Point类太简单而不能说明该问题。我们假设需要在Point类中增加一个成员函数isSamePoint()。该函数将目标对象的坐标与参数对象的坐标进行比较，如果相同则返回真；否则返回假。

```
bool Point::isSamePoint(const Point& p) const    // compare data
{ return x==p.x && y==p.y; }
```

从某种意义而言，访问另一个实例（在本例中指的是isSamePoint()中的p）是发生在目标对象（本例中是Point类型的对象）的类作用域内的，因此是允许的。

提示　当类方法的参数是类的组件类型时，该方法不能访问这个参数的私有组件，而必须使用参数的成员函数来访问。当类方法的参数与该类的类型相同时，该方法可以不需要调用访问函数而直接对参数的私有组件进行访问。使用访问函数语法上是正确的，但是并不推荐这样做。

在对象外不能访问其私有成员。为了与这个原则保持一致，同一个类的另一个对象来访问该对象的私有成员时，必须使用访问函数。该访问函数应按以下方式来编写：

```
bool Point::isSamePoint(const Point& pt) const   // compare data
  { int x1, y1;
    pt.get(x1,y1);     // overkill: access through a member function
    bool answer = (x==x1 && y==y1);
    return answer; }
```

为了避免使用这些不必要的、糟糕的代码，C++允许在访问权限上存在一点不一致。这个版本的isSamePoint()函数在语法上是正确的。如果以代码语句的行数来衡量程序的质量，程序员可以写出如上形式的大量代码。但这些代码是否可以提高应用程序的质量则是一个大问题。

12.2　复合对象的初始化

初始化是程序设计中一个重要方面。不能正确地初始化可计算对象是程序设计中常见的错误之一。C++为程序员提供了大量的初始化程序组件的方法与技术。

我们在第3章介绍定义标量变量的语法时，讨论了为这些变量设置初始值的语法。在第5章描述C++聚集语法时，讨论了数组、结构、枚举、联合、位域等程序组件的初始化问题。在第9章介绍C++类的构造语法时，也讨论了对象的初始化。

这不是巧合。对象的初始化是C++程序员主要关心的问题之一。它也涉及到把任务推向服务器类从而将客户代码从服务器类设计的细节中解放出来。我们已经介绍了复合类的语法，下面将讨论复合对象的初始化问题。在以后学习继承时也是这样：先讨论继承的语法，再分

析如何去初始化派生对象。

正如我们在第9章所提到的，在C++中定义程序变量的语法与定义类数据成员的语法是相同的。下面这行语句既可以出现在函数或块的作用域内，也可以出现在类作用域内。

```
int x,y;          // can be in a function or block; can be in a class
```

但是，只有在函数或块的可执行代码中才能在定义的同时进行初始化。

```
int x=100, y=100;  // can be in a function or block, not in a class
```

因此，不能按照初始化C++变量的语法在类定义中对其数据成员进行初始化。Java程序员可以这样做，但C++程序员则不能这样做。

```
class Point {
  int x=100, y=100;          // illegal syntax for initialization
public:
  Point (int a, int b)       // appropriate means of initialization
    { x = a;  y = b; }
     . . . } ;               // the rest of class Point
```

在复合类的函数和数据成员中定义对象实例的语法也是这样。下面这条语句既可出现在函数体中，也可出现在类定义中。

```
Point pt1, pt2;          // can be in a function or block or a class
```

但是，只有在函数体或块作用域中定义对象实例时，才允许为初始化提供参数。

```
Point pt1(20,40),pt2(70,90);      // OK in function/block, not in a class
```

这样，复合类的数据成员不能在类定义中用初始化组件类对象的语法来进行初始化。在下一个例子中，我们试图用初始化Point变量的语法来对Rectangle类的Point组件进行初始化，但编译程序不接受这种语法。

```
class Rectangle {                  // incorrect class specification
  Point pt1(20,40);                // legal in client code, illegal here
  Point pt2(70,90);                // same problem
  int thickness = 1;               // same problem
public:
  Rectangle (const Point& p1, const Point& p2, int width = 1);
  void move(int a, int b);
   . . . . } ;                     // the rest of class Rectangle
```

C++提供了两种初始化复合类组件的途径：其中一个方法是在复合类的构造函数体内进行赋值。该赋值函数可为数据成员赋值，无论这些数据成员是内部数据类型的聚集，还是内部数据类型的简单组件。

```
Rectangle::Rectangle (const Point& p1,const Point& p2,int width)
 { pt1 = p1;  pt2 = p2;      // give values to aggregate components
   thickness = width; }      // give values to built-in components
```

另外一种方法是使用一个成员初始化列表来调用类组件的构造函数。在这个例子中，成员初始化列表调用一个Point构造函数以初始化Rectangle类的Point组件。

```
Rectangle::Rectangle (const Point& p1, const Point& p2, int width)
 : pt1(p1), pt2(p2)          // call constructors for components
 { thickness = width; }      // give values to built-in components
```

警告　在C++中，定义变量和对象的语法与定义类成员的语法是一样的。但在定义类成员时不能使用初始化变量和对象的语法，但可以使用缺省的构造函数，或者使用成员初始化列表。

下面我们将首先讨论构造函数体的使用细节，然后介绍成员初始化列表的使用。

12.2.1　使用组件的缺省构造函数

在第9章，我们介绍过当创建一个C++对象时，其数据成员被分配内存。如果忽略与系统相关的为了对齐数值而需要的额外空间，我们可以假设分配给一个对象的内存是其数据成员大小的总和。

前面曾经说过，创建一个对象时，其数据成员在构造函数调用时被初始化，我们需要强调这一点。虽然认为构造函数是在创建对象的同时被调用的更加便于理解，但更确切的说法是：构造函数应是在对象的所有数据成员被构造之后调用的。

然而我们也承认，对于只有内部数据类型的非复合域的类而言，同时调用与之后才调用构造函数之间的区别并不重要。这类似于初始化与赋值之间的区别。对于内部数据类型的非复合变量，这个区别是相当小的。正如在第11章所讨论的，对于动态处理堆内存的类而言，这个区别就变得非常重要了。不能区分它们将影响程序的性能和完整性。

类似地，如果对象的组件是程序员定义的类的对象，上述两者之间的区别也很重要。在本节，我们将仔细讨论创建一个复合对象的过程。

简而言之，在C++中创建一个对象时，其数据成员被分配内存，接着执行构造函数体。这意味着对象的构造函数只在其所有数据成员创建之后才调用。

这个过程的一个重要特点是数据成员的创建次序是按其在类标识中出现的次序。当该过程结束后，复合类型的对象对于内存而言是该对象的全部组件的集合。

这就意味着在类中改变数据成员的次序将会影响类的属性，但只有当数据成员相互依赖时才会如此。例如，一个数据成员可能表示另一个数据成员中组件的个数。稍后，我们将会看到数据依赖的例子。

当数据成员一个接一个地创建后，如果存在属于内部数据类型的域，那么它们或者没有初始化（如果对象是在堆中或在栈中创建的）或者被置为空（对于那些作为全局或静态对象而创建的对象而言）。如果对象需要在一个内部数据类型的域中存储某个特定的值，则要稍后即在调用构造函数时进行处理。例如，`Rectangle`类对象的`thickness`域被设置为构造函数的参数`width`的值。

如果对象是一个复合对象，其数据成员是其他类的对象时，情况又会如何呢？前面所提到的“数据成员的创建次序是按类标识中的次序”将有所提示。对象创建过程是一个递归过程，一个数据成员创建之后再调用其类的构造函数。

我们曾经提到过，任何C++对象的创建都需要调用构造函数来分配内存。如果复合对象的某个域是程序员定义类型（结构或类），其构造函数在为该域分配内存之后创建下一个域之前立即调用。只有当复合对象的所有域都成功创建并初始化后，才执行复合类的构造函数体。

因此，当创建`Rectangle`类的一个对象时，事件发生的次序如下：

1) 创建`Point`类的数据成员`pt1`。

2) 创建`Point`类的数据成员`pt2`。

3) 创建int类型的数据成员thickness。

4) 执行Rectangle类的构造函数体。

在构造Rectangle对象过程中，每个Point类创建的数据成员，其次序如下：

1) 创建int类型的数据成员x。

2) 创建int类型的数据成员y。

3) 执行Point类的构造函数体。

我们可以看到，在执行Rectangle类的构造函数体之前，调用了两次Point类的构造函数：第一次初始化数据成员pt1域，第二次初始化数据成员pt2域。

注意在撤销复合对象时，内存管理和函数调用的过程要反过来进行。首先调用复合类的析构函数，然后再回收所有的内存。当析构函数终止时，数据成员撤销的次序与其创建次序相反。撤销每个数据成员时，其过程是递归进行的。首先调用组件析构函数，然后撤销组件内存。在组件析构函数终止后，撤销组件数据成员（从组件类定义中的最后一个到第一个依次进行撤销）。

这样，撤销Rectangle对象的事件序列是创建该对象事件序列的一个镜像。

1) 执行Rectangle的析构函数。

2) 撤销数据成员thickness。

3) 为数据成员pt2执行Point类析构函数。

4) 撤销数据成员pt2（先撤销它的y域，再撤销它的x域）。

5) 为数据成员pt1执行Point析构函数。

6) 撤销数据成员pt1（先撤销它的y域，再撤销它的x域）。

在描述Rectangle对象创建过程时，我们对调用Rectangle构造函数十分肯定，因为Rectangle类只有一个构造函数。但是当我们指出Point类的构造函数会被执行时就不那么肯定。当创建Point数据域对象时调用什么构造函数呢？

对于任何一个C++函数，其答案依赖于客户在构造函数调用中所提供的参数数目。有些程序员容易犯概念性错误，他们认为如果没有提供参数则不会调用构造函数。实际并非如此，如果没有提供参数，就会调用无参数的构造函数，即调用缺省的构造函数。当提供一个参数时，就调用仅带有一个同样类型参数的构造函数。依次类推。如果所需的构造函数无效怎么办？与使用不正确的参数调用任何C++函数一样，它意味着这个函数调用（试图创建一个对象）将产生一个语法错误。

在下面这段客户代码中，我们可以看到该代码为调用Rectangle构造函数传递参数。但没有参数传递给Point构造函数。

```
Point p1(20,40), p2(70,90);    // top-left and bottom-right corners
Rectangle rec(p1,p2,4);        // this is a syntax error
```

这意味着在创建复合对象过程中，创建组件类型的数据成员时，调用了组件类的缺省构造函数。

在这个例子中，Point类没有缺省的构造函数。这是一个不好的消息。但是当类没有缺省的构造函数时，编译程序会提供一个构造函数。该构造函数什么工作也不做，只是让客户代码在没有参数传递给构造函数时也可以创建对象。这是个好消息。然而，如果类有非缺省的构造函数（Point类有一个），编译程序将不使用它自己缺省的构造函数。这样，定义复合

类对象会产生语法错误。因此，上面程序段的最后一行是错误的。

这是一个类与类链接（Point与Rectangle）的例子，程序员往往对此模糊不清。有时候，在定义Rectangle类时会出错，因为要调用Point类中并不存在的构造函数。但是，在编译Point类甚至Rectangle类时都不会出现语法错误。

出现错误的逻辑根源在于组件类Point的设计上。因为该类没有缺省的构造函数。但是，该逻辑错误并不在组件Point类的设计中出现，甚至也不在复合类Rectangle的设计中出现，而是出现在Rectangle类的客户代码中，当要去实例化复合类对象时。该代码的位置与错误的根源所在的代码相距甚远。除非客户代码试图去定义Rectangle对象，否则该程序不会出现语法错误。

为了修正这种错误，可以在Point类中添加一个缺省的构造函数。这将消除上面那段代码中的语法错误。

```
class Point {
    int x, y;                    // private coordinates
  public:
    Point ()
      { x=0;  y=0; }             // default constructor
    Point (int a, int b)         // general constructor
      { x = a;  y = b; }
   . . . } ;                     // the rest of Point class
```

另一个办法是在通用的构造函数中添加缺省的参数值，这样，该构造函数既可以作为一个缺省的构造函数，也可以作为一个转换构造函数。

```
class Point {
  int x, y;                    // private coordinates
public:
  Point (int a=0, int b=0)
  { x = a;  y = b; }           // default, conversion, general constructor
   . . . } ;                   // the rest of Point class
```

作为总结，我们再来看一下Rectangle对象的创建步骤，图12-1表示了下面这段客户代码的执行情况。

```
Point p1(20,40), p2(70,90);  // top-left and bottom-right corners
Rectangle rec(p1,p2,4);      // OK if Point has default constructor
```

首先，给对象pt1分配内存，调用缺省的Point构造函数，它将pt1.x和pt1.y的值设置为0。接着，给对象pt2分配内存，调用缺省的Point构造函数，它将pt2.x和pt2.y的值设置为0。接着给数据成员thickness分配内存，但没有对它初始化。随后Rectangle构造函数被调用，执行Rectangle构造函数体时，首先将参数p1的内容拷贝给数据成员pt1，接着把参数p2的内容拷贝给pt2，然后将thickness的值设为4。

结果，创建了Rectangle对象，pt1.x设置为20，pt1.y设置为40，pt2.x设置为70，pt2.y设置为90。

能理解这个过程吗？调用Point缺省构造函数给pt1和pt2所设置的值并不长久，在调用Rectangle构造函数时p1和p2的值覆盖了它们的值。两个Point数据成员所调用的缺省构造函数徒劳无功。这是否有些让人生气？因为Rectangle对象的创建过程中要浪费数毫秒。其实没什么，这意味着我们已经明白了为错误的C++编程要付出怎样的代价。

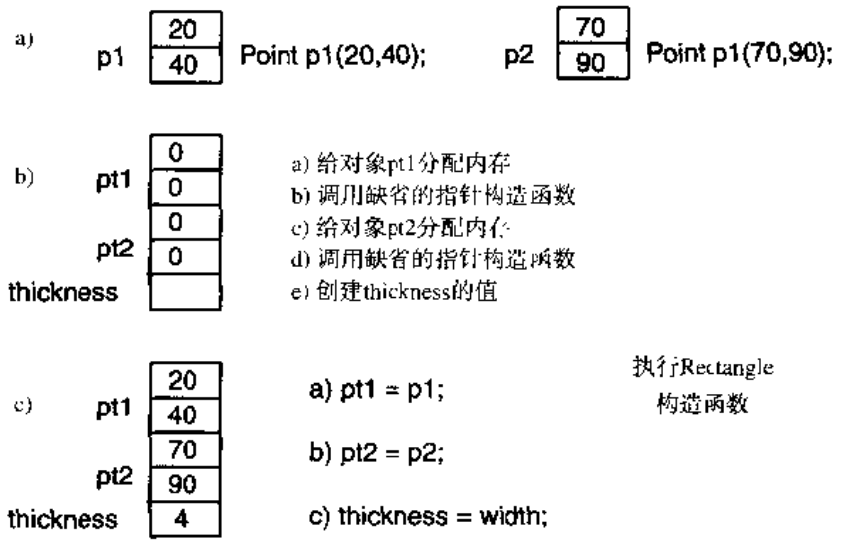


图12-1 调用Point缺省的构造函数来创建Rectangle对象的步骤

注意 在C++中，对象的实例化总会涉及到一个函数调用，即调用类的构造函数。复合对象的实例化将涉及不止一个函数调用：每个数据成员被创建之后都要立即调用一个构造函数。要学会阅读C++程序中的函数调用。

我们多次强调的是，当必须在程序的可读性和性能之间进行选择时，我们推荐选择可读性。但是，即使没有充分的理由浪费执行时间，也没有办法两者兼得。一个C++程序员应该培养这种能力，在阅读C++代码时留意这些多余的函数调用。作为C++程序员应该知道如何尽量避免这种浪费。

程序12-1实现了复合类Rectangle和组件类Point，以便测试。为了帮助分析其结果，我们在Point类中用调试消息增加了一个拷贝构造函数和重载的赋值运算符函数，来跟踪Rectangle复合对象的创建过程。该程序的执行结果如图12-2所示。

程序12-1 复合对象创建过程中调用了多余的构造函数

```
#include <iostream>
using namespace std;

class Point {
  private:
    int x, y;                                   // private coordinates
  public:
    Point (int a=0, int b=0)                    // general constructor
    { x = a;  y = b;
      cout << " Created: x= " << x << " y=" << y << endl; }
    Point (const Point& pt)                     // copy constructor
    { x = pt.x;  y = pt.y;
      cout << " Copied: x= " << x << " y=" << y << endl; }
    void operator = (const Point& pt)           // assignment operator
    { x = pt.x;  y = pt.y;
      cout << " Assigned: x= " << x << " y=" << y << endl; }
    void set (int a, int b)                     // modifier function
      { x = a;  y = b; }
    void move (int a, int b)                    // modifier function
```

```
    { x += a;  y += b; }
  void get (int& a, int& b) const          // selector function
    { a = x;  b = y; } } ;

class Rectangle {
  Point pt1, pt2;                          // top-left, bottom-right corner points
  int thickness;                           // thickness of the rectangle border
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                 // move both points
  bool pointIn(const Point& pt) const;     // point in rectangle?
  } ;
Rectangle::Rectangle(const Point& p1,const Point& p2,int width)
  { pt1 = p1; pt2 = p2; thickness = width; }    // set data members

void Rectangle::move(int a, int b)
  { pt1.move(a,b);  pt2.move(a,b); }           // pass buck to Point

bool Rectangle::pointIn(const Point& pt) const       // is point in?
{ int x,y,x1,y1,x2,y2;                         // coordinates of pt and corners
  pt.get(x,y);                                 // get parameter's coordinates
  pt1.get(x1,y1);  pt2.get(x2,y2);             // get data from corners
  bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }

int main()
{
  Point p1(20,40), p2(70,90);                  // top-left, bottom-right corners
  Point point(100,120);                        // point to catch with the rectangle
  Rectangle rec(p1,p2,4);                      // wasted constructor calls
  point.move(-25,-15);                         // move the point around the screen
  rec.move(10,20);                             // 10 pixels to right, 20 pixels down
  if (rec.pointIn(point)) cout << " Point is in\n";     // point in?
  return 0;
  }
```

图12-2的前三条Created消息反映了Point对象p1、p2和point的创建。后两条Created消息描述了Rectangle对象的创建：前一条消息描述了数据成员pt1的创建及对Point缺省构造函数的调用，后一条消息描述了数据成员pt2的创建及对Point缺省构造函数的调用。两条Assigned消息刻画了对象创建完成之后Rectangle构造函数的执行过程：前一条消息对应于构造函数体中第一个赋值，后一条消息对应于构造函数体中第二个赋值。

```
Created: x= 20 y=40
Created: x= 70 y=90
Created: x= 100 y=120
Created: x= 0 y=0
Created: x= 0 y=0
Assigned: x= 20 y=40
Assigned: x= 70 y=90
Point is in
```

图12-2　程序12-1的输出结果

这是C++中复合对象创建的典型图示。对于较大的复合对象，创建过程所涉及的问题可能

更多，多余的构造函数也越多。当然，我们无法在各个数据成员创建之后就立即去掉这些构造函数调用。这是C++中的一条严格规则：任何对象创建时必须紧接着调用构造函数。但我们可以（也应该）试着去调用这样的一个构造函数，它的工作成果能保持到复合对象构造函数执行之后。

12.2.2 使用成员的初始化列表

在C++的复合类构造函数中，可以使用成员初始化列表，或初始化程序列表来避免上面所出现的多余的函数调用。其语法有点特殊：使用构造函数头和构造函数体之间的位置。下面是一个成员初始化列表的例子：

```
class Rectangle {
  Point pt1, pt2;                                  // Top-Left, Bottom-Right;
  int thickness;
public:
  Rectangle (const Point& p1, const Point& p2, int w = 1);
  . . . . } ;                                      // the rest of class Rectangle

Rectangle::Rectangle(const Point& p1,
            const Point& p2,int w) : pt1(p1),pt2(p2)
{ thickness = w; }                                 // this is much better!
```

初始化程序列表位于构造函数参数列表的闭圆括号与构造函数体的开花括号之间。它以冒号开始，列举数据成员的名字（而不是类型）。在每个数据成员名字后有一个括号、括号中列出了用来初始化对象数据成员的相应参数值，数据成员名之间以逗号隔开。初始化列表没有终止符，它在构造函数体的开花括号处结束。初始化程序列表中的每一项都形如定义变量时的构造函数调用，如pt1(p1)。

注意初始化程序列表的语法只适用于构造函数实现，它不影响构造函数原型的编写形式。不要将它与参数的缺省值相混淆。缺省值只适用于原型，它不影响构造函数实现的编写形式。

初始化程序列表将强迫编译程序调用数据成员的具有合适参数个数的构造函数。在为数据成员分配内存之后、在复合类构造函数体执行之前调用这些构造函数。因此，组件数据成员在调用复合类构造函数时已被初始化了。如果需要，可以在复合类的构造函数体中使用这些组件数据成员。

实际上，任何数据（包括内部数据类型的数据）都可在列表中初始化。下面是Rectangle类的构造函数，所有的数据成员都在初始化列表中被初始化。

```
Rectangle::Rectangle(const Point& p1, const Point& p2, int w)
   : thickness(w), pt1(p1), pt2(p2)        // on the line by itself
  { }                                      // empty body: a popular C++ idiom
```

如上所示，初始化程序列表可独占一行，这是该语法常见的使用情况。使用这种扩展了的初始化列表会产生一种奇怪的现象。即在构造函数体被执行前就没有什么其他工作需要处理了。因此，构造函数体是空的。但它必须存在，因为函数体不能被遗漏。在初始化程序列表中初始化原子数据类型实际上没有什么好处，但因为某种原因，C++程序员都比较喜欢使用空的构造函数体。

顺便指出，以上的初始化程序列表可能会使人认为，thickness数据成员在pt1和pt2数据成员之前初始化。事实上并非如此。在初始化程序列表中指定组件的顺序不会产生任何

影响。它们按照类定义中的先后次序进行初始化。

使用初始化程序列表来执行下面这段程序时，创建Rectangle对象的事件序列如图12-3所示。

```
Point p1(20,40), p2(70,90);          // top-left and bottom-right corners
Rectangle rec(p1,p2,4);              // no need for Point default constructor
```

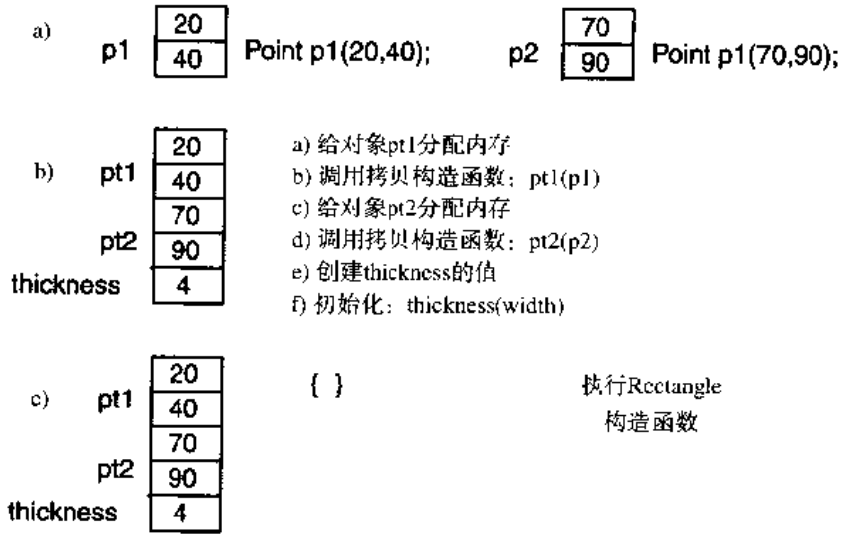


图12-3　用成员初始化程序列表创建Rectangle对象的步骤

在为点p1和p2分配内存后，Rectangle对象rec被构造。首先，创建Point数据成员pt1，然后以对象p1为参数为这个数据成员调用Point类的拷贝构造函数。结果，数据成员pt1被初始化：x为20，y为40。接着，创建Point数据成员pt2，然后以对象p2为参数为这个数据成员调用Point类的拷贝构造函数。结果，数据成员pt2被初始化：x为70，y为90。随后，创建数据成员thickness并把它初始化为4。大功告成！这样，就不用白白调用Point类的缺省构造函数了。

程序12-2与程序12-1大致相同，只是其复合类Rectangle的设计不同而已。程序12-1中实现了通用的构造函数，在程序12-2中则实现了带有成员初始化列表的Rectangle类构造函数。该程序的执行结果如图12-4所示。

程序12-2　创建复合对象时没有调用多余的构造函数

```
#include <iostream>
using namespace std;

class Point {
  private:
    int x, y;                              // private coordinates
  public:
    Point (int a=0, int b=0)               // general constructor
    { x = a;  y = b;
      cout << " Created: x= " << x << " y=" << y << endl; }
    Point (const Point& pt)                // copy constructor
    { x = pt.x;  y = pt.y;
      cout << " Copied: x= " << x << " y=" << y << endl; }
    void operator = (const Point& pt)      // assignment operator
    { x = pt.x;  y = pt.y;
```

```
    cout << " Assigned: x= " << x << " y=" << y << endl; }
  void set (int a, int b)              // modifier function
    { x = a;  y = b; }
  void move (int a, int b)             // modifier function
    { x += a;  y += b; }
  void get (int& a, int& b) const      // selector function
    { a = x;  b = y; } } ;

class Rectangle {
  Point pt1, pt2;                      // top-left, bottom-right corner points
  int thickness;                       // thickness of the rectangle border
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);             // move both points
  bool pointIn(const Point& pt) const; // point in rectangle?
   } ;

Rectangle::Rectangle(const Point& p1, const Point& p2,int w)
: thickness(w), pt1(p1), pt2(p2)       // initialization list
  { }                                  // empty member body

void Rectangle::move(int a, int b)
  { pt1.move(a,b);  pt2.move(a,b); }   // pass buck to Point

bool Rectangle::pointIn(const Point& pt) const     // is point in?
 { int x,y,x1,y1,x2,y2;                // coordinates of pt and corners
   pt.get(x,y);                        // get parameter's coordinates
   pt1.get(x1,y1);  pt2.get(x2,y2);    // get data from corners
   bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
   bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
   return (xIsBetweenBorders && yIsBetweenBorders); }

int main()
{ Point p1(20,40), p2(70,90);          // top-left, bottom-right corners
  Point point(100,120);                // point to catch with the rectangle
  Rectangle rec(p1,p2,4);              // NO wasted constructor calls
  point.move(-25,-15);                 // move the point around the screen
  rec.move(10,20);                     // 10 pixels to right, 20 pixels down
  if (rec.pointIn(point)) cout << " Point is in\n";  // is point?
  return 0;
  }
```

```
Created: x= 20 y=40
Created: x= 70 y=90
Created: x= 100 y=120
Copied: x= 20 y=40
Copied: x= 70 y=90
Point is in
```

图12-4 程序12-2的输出结果

图12-4中的前三条Created消息与图12-2中的前三条消息相同：它们反映了main()函数中三个Point对象的创建过程。紧接着的两条Copied消息反映了Rectangle对象的创建过程：前一条消息在创建数据成员pt1之后调用Point类拷贝构造函数时出现，后一条消息在创建数据成员pt2之后调用Point类拷贝构造函数时出现。我们可以看到，这里调用的是

Point类的拷贝构造函数，而不是缺省的构造函数。而且它的结果是持续的。在Rectangle构造函数调用的过程中，没有调用Point类的赋值运算符函数，Rectangle类的构造函数体是空的。一切都让人满意！

在这些例子中，复合类仅有一个构造函数。如果复合类有几个构造函数，每个构造函数都会遵循同样的逻辑。在创建一个复合对象时，先创建其数据成员。最后再根据客户代码为复合对象提供的参数个数与参数类型，调用合适的复合类构造函数。如果复合类有所需的构造函数，则调用这个构造函数；如果没有，对象实例化时将产生一个语法错误。如果最终调用的构造函数没有成员初始化列表，那么每个数据成员创建之后将调用缺省的组件类构造函数。如果复合类构造函数有成员初始化列表，列表上的每个元素将调用相应的组件类构造函数。

由不同构造函数实现的初始化程序列表可以完全不同。下面是Rectangle类的另一版本，它重载了3个构造函数：其中一个为前面例子中的通用构造函数；一个是带有4个参数的通用构造函数，这4个参数表示两个点的坐标；另一个是缺省的构造函数。每个构造函数都有各自的初始化列表。这些初始化列表可以互不相同。

```
class Rectangle {
  Point pt1, pt2;                    // Top-Left, Bottom-Right;
  int thickness;
public:
  Rectangle (const Point& p1, const Point& p2, int w = 1);
  Rectangle (int x1, int y1, int x2, int y2);
  Rectangle ();
   . . . . } ;                       // the rest of class Rectangle

Rectangle::Rectangle(const Point& p1, const Point& p2,int w)
  : thickness(w), pt1(p1), pt2(p2) { }

Rectangle::Rectangle (int x1, int y1, int x2, int y2)
  : pt1(x1,y1), pt2(x2,y2), thickness (1) { }

Rectangle::Rectangle () : pt1(0,0), pt2(100,100), thickness(1)
  { }
```

第一个构造函数演示了如何将复合类构造函数的参数作为组件类构造函数的参数。在这个例子中，Point类型的参数p1作为拷贝构造函数的参数，用来初始化数据成员pt1；Point类型的参数p2作为拷贝构造函数的参数，用来初始化数据成员pt2。最后一个构造函数的参数用来初始化整数类型的数据成员，即thickness。

第二个构造函数演示了成员初始化列表的使用不受客户代码所提供的参数的限制。这里客户代码只提供了调用Point通用构造函数的数据。初始化数据成员thickness的值被定义为一个常量字符。这完全是可以的。

注意与缺省值类似，在这个成员初始化列表中使用字面值表示将任务从客户推给服务器类。在该设计版本中，是Rectangle类的编写者指定了矩形线的宽度为1。我们可以用另一个参数来代替字符值，如下所示。

```
Rectangle::Rectangle (int x1, int y1, int x2, int y2, int width)
  : pt1(x1,y1), pt2(x2,y2), thickness (width) { }
```

这样，指定线宽度为1的任务就变成客户代码的了。

```
Rectangle r(20,40,70,90,1);    // responsibility is pushed up client
```

Rectangle类的第三个构造函数是缺省的构造函数，在列表参数中也使用了常量字符。使用该构造函数时，每个Rectangle对象将左上角初始化为原点坐标、右下角为坐标（100，100）。缺省构造函数没有从客户代码接受任何信息。因此，由缺省构造函数初始化的所有对象都会具有相同的状态，这是很自然的事情。

由于Point类提供了一个缺省的构造函数（坐标值为0），该构造函数的Rectangle初始化列表形式如下：

```
Rectangle::Rectangle () : pt1(), pt2(100,100), thickness(1) { }
```

如果忽略了初始化列表中的数据成员pt1，结果会怎样呢？成员初始化列表的目的是避免在复合类构造函数调用之前去调用组件类的缺省构造函数。成员初始化列表将调用列表中指定的构造函数来代替组件类的缺省构造函数的调用。

因此，在Rectangle类构造函数的这个例子中，我们试图避免为数据成员pt1调用Point的缺省构造函数，而代之为数据成员pt1调用其缺省构造函数。这样，成员初始化列表中的缺省构造函数调用就可以省略。这不会改变创建复合类对象的事件次序。这个Rectangle构造函数的形式如下：

```
Rectangle::Rectangle () : pt2(100,100), thickness(1) { }     // same
```

这个初始化程序列表语法有点儿怪，与我们在前面所见到的不一样。许多程序员学习和使用这种语法都会有困难。但它是用C++类编程的一个重要部分。

提示 要掌握初始化程序列表的语法，因为它是非常有用的。使用初始化列表，可以在初始化复合对象的组件时不调用多余的构造函数。它在C++代码中十分常见。稍后将讨论它在继承中的使用。不使用初始化列表，我们将无法写出有意义的C++程序。越早学习初始化列表越好。

可以逐步地学习这个语法。初始化列表的使用是可选的。使用初始化列表的第一个目的是避免当组件的缺省构造函数不存在时出现的语法错误。第二个目的是避免当组件状态在复合类构造函数中被重新设置后出现的性能上的不良影响。也就是说，其目的是防止语法错误并改善程序性能。语法错误可通过提供组件类的缺省构造函数来避免。性能问题则可能根本不那么重要。

但对于常量数据成员和引用数据成员而言，则必须使用初始化列表。

12.3 具有特殊属性的数据成员

以前，我们可能从未考虑过使用常量与引用数据成员，在前面讨论软件工程的有关问题时也未涉及该问题。在第8章的最后，我们列举了与C++类相关的问题：绑定数据和操作，引入数据成员和成员函数名字的类作用域，类成员的访问控制，将任务从客户推到服务器等。

所有这些都是合法的目标，且与面向对象程序设计的基本原则一致。第9~11章讨论了使用类的其他一些情况，例如对象的自动初始化、管理堆内存、以同样方式处理对象与内部数据类型的变量。但是除了C++类允许我们做的其他操作之外，我们还没有提到：可以将数据成员定义为常量或引用。可以说这一点是学习了所有这些知识之后获得的惊喜。

如果对本部分的内容足够熟悉，可以跳过本节和下一节，有必要时再回过头来学习。

12.3.1 常量数据成员

常量数据成员的思想很简单。一个C++类将相关的数据成员和函数捆绑在一起，函数代表客户代码访问数据成员。客户代码常常需要改变对象的状态（如账户收支情况、职员地址、电影租费总额）。但对象的有些属性是不能被修改的（如账号、雇员聘用的日期、支付给电影销售商的费用）。

类设计者知道类的成员函数不能修改某个数据成员的值，如账号等。维护人员则必须通过研究类成员函数和友元函数来推测出这一点。因此，有必要花些精力在类代码中显式地说明那些设计者在设计时了解的内容：这个数据成员的值不能被任何成员函数或者友元函数修改。这是const关键字的另外一个工作。

如何初始化这样的常量数据成员呢？如果在类构造函数中初始化，似乎太迟了。回顾一下图12-1中所描述的，类构造函数是在对象已创建之后才被调用的。在类构造函数体中是一个赋值操作，而不是初始化。不允许对常量数据成员赋值，它必须在创建之后马上被初始化。因此，C++要求在初始化程序列表中包含常量数据成员的名字，而且在任何情况下对该数据成员进行赋值都将标注为语法错误，即使在构造函数中也不允许对它赋值。

常量数据成员可以是内部数据类型，也可以是程序员定义的数据类型。这与我们正讨论的问题没有关系，我们讨论的是要在创建该数据成员之后立即对它初始化，初始化一定要在调用构造函数之前完成，而不是延后。

作为使用常量数据成员的例子，我们在Rectangle类中增加了一个数据成员来描述矩形区域单元的权。由于制作矩形的材料在矩形的生命期内保持不变，该数据成员的值在Rectangle对象创建之后是不会改变的。为了避免让维护人员搜索所有的类成员函数和友元函数来确认这一点，应该将weight数据成员定义为常量。这样，该数据成员应该在Rectangle类构造函数的成员初始化列表中初始化。

```
class Rectangle {
  Point pt1;
  Point pt2;
  int thickness;
  const double weight;                                  // weight of one unit of area
public:
  Rectangle (const Point& p1, const Point& p2, double wt, int width = 1);
  void move(int a, int b);
  void setThickness(int w=1);
  int pointIn(const Point& pt) const;
   . . . . } ;                                          // the rest of class Rectangle

Rectangle::Rectangle(const Point& p1, const Point& p2, double wt, int width)
   : pt1(p1), pt2(p2), weight(wt)                       // weight is not optional here
{ thickness = width; }
```

注意，我们将表示权的参数添加在构造函数参数列表的中间，而不是最后。还记得那条只能对最右边的参数设置缺省值的规则吗？如果我们把第4个参数添加到参数列表的最后（如下行所示），则违背了该原则。

```
Rectangle(const Point& p1, const Point& p2, int width=1, double wt);   // wrong
```

Rectangle的客户代码改动不大。

```
Point p1(20,40), p2(70,90);          // top-left, bottom-right corners
Point point(100,120);                // point to catch with the rectangle
Rectangle rec(p1,p2,0.01,4);         // supported by the initializer list
rec.setThickness();                  // line width is 1 pixel (default)
point.move(-25,-15);                 // move the point around the screen
rec.move(10,20);                     // 10 pixels to right, 20 pixels down
if (rec.pointIn(point)) cout << "Point is in\n";    // is point in?
p1.move(30,35);                      // does the rectangle object change?
```

与const关键字的其他用法类似，在定义常量数据成员时使用const限制符可以让程序代码易读。理论上而言，如果一个数据成员没有使用const关键字，这意味着该数据成员的值在对象的生命期内可以被所属类的某个函数改变。但我们发现极少有程序员使用常量数据成员，因此，在数据成员的定义中即使没有发现const关键字，也不能认为可以改变该数据成员的值。这只能表明程序员忙着考虑类设计中的其他方面，而忽略了尽量将其设计意图传达给维护人员。

12.3.2 引用数据成员

下面我们讨论一个将对象引用作为其他对象的数据成员的例子。当几个客户对象与同一服务器对象相关时，这些对象之间的关联可由对象引用来实现。

在本章前面的几个例子中，每个Rectangle对象拥有其角坐标点的副本。如果移动了p1（如前一个例子的最后一行），Rectangle对象的rec没有变化。对许多应用程序而言，这正是对象应该具有的行为特点。在第11章中，这是由值语义来实现的。

对其他应用程序而言，可能希望几个矩形共享某些点。这样，如果客户代码移动了矩形角坐标上的点，矩形也应随之改变。在第11章中，我们是用引用语义来实现的。用面向对象方法设计的许多程序使用引用语义来实现对象之间的关系。例如，账户所有者数据（姓名、地址、社会保险号等）可能是账户类的一部分。如果应用程序要用到所有者，所有者数据可被合并为一个类，并可以将所有者对象当做账户类的一个数据成员。如果一个所有者有几个账户，应用程序希望为这些账户只使用一个所有者对象。这样，对所有者数据的修改将会自动传播给所有的账户。

可以使用引用数据成员支持上述客户代码的功能。（注意我们一直在强调服务器类是根据客户代码的需要而设计的，而不是根据通用性或性能来设计。）这些引用可指向复合对象之外的对象，客户代码可不经过复合对象的同意对这些外部对象进行修改。

正如我们在前面曾经提到过的，C++中的所有引用都是常量。引用初始化后就不能修改。这样，引用数据成员（与常量数据成员类似）只能在成员初始化列表中初始化，而不允许在构造函数体内初始化。这样太迟了，因为该构造函数的调用是在创建了所有的数据成员并调用了这些数据成员的构造函数之后。这个新的Rectangle类设计与前面版本大致相同，惟一区别是这个Rectangle类数据成员定义中Point类型后的两个&符号。

```
class Rectangle {
    Point& pt1;                      // points can be shared with other shapes
    Point& pt2;
    int thickness;
    const int weight;                // weight of one unit of area
  public:
    Rectangle (const Point& p1, const Point& p2,
```

```
                    int wid = 1, int wt = 1);
    void move(int a, int b);
    void setThickness(int w=1);
    int pointIn(const Point &pt) const;
     . . . . } ;                        // the rest of Rectangle class

Rectangle::Rectangle(const Point& p1, const Point& p2, int width, int wt)
    : pt1(p1), pt2(p2), weight(wt)    // this is not optional here
{ thickness = width; }                 // same constructor as above
```

由于所有的引用都是const，就没有必要再来标记该属性。在Rectangle类中，pt1和pt2是常量引用，它们不能抛弃正指向的对象而去指向别的对象。但这些对象本身不是常量，其内容可被改变。

与指针所指向的对象相似，引用所指向的对象也可定义为常量。这样，不仅pt1和pt2都指向相同的Point对象，不能再指向其他的Point对象；而且这些对象的状态也不能被改变。

```
class Rectangle {
  const Point& pt1;        // points can be shared with other shapes
  const Point& pt2;        // points cannot change their coordinates
  . . . . } ;              // the rest of Rectangle class
```

从语法上讲，其要求是相同的：这些数据成员必须在成员初始化列表中被初始化。从语义上讲，上述方案意义不大。如果角坐标上的点是常量，那么与其他Rectangle对象共享它们就没有什么优势。可将这些点作为常量数据成员。

```
class Rectangle {
  const Point pt1;         // points are not shared with other shapes
  const Point pt2;         // points cannot change their coordinates
  . . . . } ;              // the rest of Rectangle class
```

使用常量对象的引用是一种优化技术。如果许多复合对象都拥有同一组件对象，这样可以只创建一个组件对象，并在所有复合对象中建立该对象的引用。

创建Rectangle类的对象的过程如图12-5所示。首先创建了Point类型的对象（如图12-5a所示）；接着，创建Rectangle类对象（如图12-5b所示）；创建引用pt1和pt2，这里与前面的例子不同的是，pt1和pt2是不同的类型，我们用不同大小的区域来表示它们类型的不同。接着，执行初始化列表，将引用指向Point对象；再创建和初始化常量域weight，然后创建thickness域。最后执行Rectangle构造函数（如图12-5c所示），并给thickness域赋值。

在这个例子中，Rectangle类中的引用是常量，但这些引用所指向的Point对象不是常量。因此，这些Point对象的状态可以改变，而且与这些Point对象相关联的所有Rectangle对象在屏幕上的位置也会随之改变。但是，使用引用不允许Rectangle对象抛弃与之相关联的Point对象而使用另一Point对象。但如果Rectangle类使用Point指针代替引用时，这种情况就是允许的。

```
class Rectangle {
  Point *pt1, *pt2;                // points can be shared with other shapes
  int thickness;
  const int weight;                // weight of one unit of area
public:
```

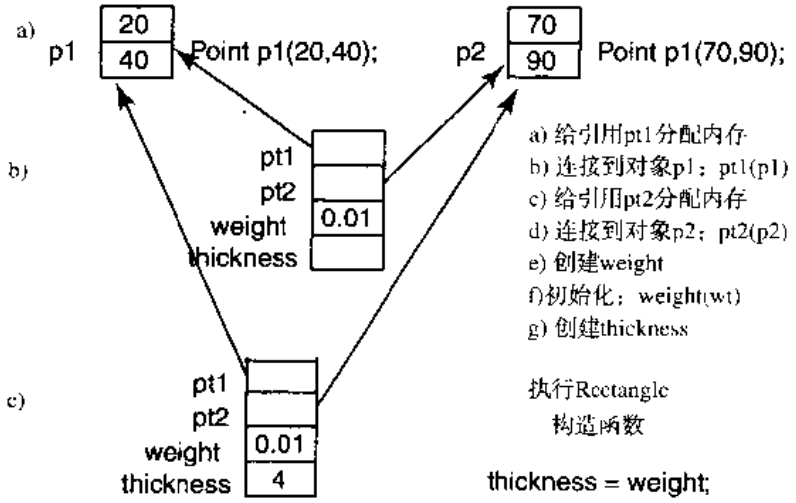


图12-5 创建Rectangle对象，使其引用指向外部对象的步骤

```
    Rectangle (const Point*, const Point*, int = 1, int = 1);
    void move(int a, int b);
    void setThickness(int w=1);
    int pointIn(const Point &pt) const;
    . . . . } ;                          // the rest of Rectangle class

Rectangle::Rectangle(const Point *p1, const Point *p2,
  int width, int wt)   : pt1(p1), pt2(p2), weight(wt)    // optional again
 { thickness = width; }                  // same constructor as above
```

由于指针在其生命期内可随时被修改，因此并不一定需要在这里使用成员初始化列表。但它是一个好的做法。

如果由指针所指向的对象在Rectangle对象的生命期内要保持不变，那么可将指针声明为常量对象的指针。

```
class Rectangle {
  const Point *pt1;       // points can be shared with other shapes
  const Point *pt2;       // point coordinates cannot change
   . . . . } ;            // the rest of Rectangle class
```

如果许多Rectangle对象与相同的Point对象关联时，该方案是一个很有用的优化策略。

不要过多地使用常量数据成员和引用数据成员。然而，只要它们反映了服务器对象和客户代码的需求，就是合法的设计工具。

12.3.3 用对象作为其类自身的数据成员

在前面几节，我们讨论了将某个类的对象（如Point）作为另一个类的数据成员（如Rectangle）。那么，一个对象可否作为其类自身的成员呢？例如，应用程序需要一些与屏幕上的几个焦点（focal points）相关联的点坐标。对于每个点，应用程序可能想要定义一个锚点（anchor point）表示Point对象的特征。

```
class Point {
    int x, y;                       // private coordinates
    Point anchor;                   // this is not allowed
```

```
  public:
    Point (int a=0, int b=0)        // versatile constructor
      { x = a;  y = b; }
. . . } ;                           // the rest of class Point
```

但这是不允许的。在本章开始，我们曾讨论过分配对象内存的事件序列。数据成员是按类规格说明中这些成员的定义次序来分配内存的（静态成员在程序开始执行时分配）。因此，创建一个Point对象时，先为x和y分配内存，然后为anchor分配内存。但anchor是Point类型，这样，先分配anchor数据成员的x和y的内存，再为anchor分配内存，依此类推。这个递归过程无法结束，因此是不允许的。

可以引用对象自身的类的对象，同样也可以使用指向同一个类的对象的指针。指针和引用表示的都是一个对象的地址，并不需要为整个对象分配内存。因此，可以比较容易地为它们在其他数据成员中分配内存。

```
class Point {
  int x, y;                         // private coordinates
  Point &anchor;                    // this is reasonable
public:
  Point (int a=0, int b=0, Point &focus)
   : anchor (focus)                 // cannot be set in constructor
    { x = a;  y = b; }
. . . } ;                           // the rest of class Point
```

我们曾提到过，一个引用数据成员不能在类的构造函数体中被初始化。因此，我们使用了作为参数传递给构造函数的Point对象，以在初始化程序列表中对引用数据成员进行初始化。这个构造函数的参数不能有const修饰符，因为anchor数据成员没有定义为常量。使用anchor引用，该Point对象可以修改作为参数传递给它的对象。因此，将参数定义为常量是一个语法错误。

注意，在这里我们给构造函数增加了一个新的参数，并将它设置为最右边的参数。这是一个比较常见的错误。该参数没有缺省值，因此应将它移到那些有缺省值的参数的左边。

```
class Point {
    int x, y;                       // private coordinates
    Point &anchor;                  // this is reasonable
  public:
    Point (Point &focus, int a=0, int b=0)      // better order
     : anchor (focus)               // cannot be set in constructor
      { x = a;  y = b;  }
. . . } ;                           // the rest of class Point
```

客户代码首先创建锚点，然后将它作为一个参数传递给即将要创建的新点的构造函数。第一个点必须是它自己的锚点。

```
Point p1(p1);       // syntax error: p1 is not defined as an argument
```

这是一个语法错误。将p1作为构造函数的参数时，p1还在创建过程中，其引用还没有定义，因此，我们要动态地分配锚点。

```
Point *p = new Point(*p,80,90);     // p has no value yet
Point p1(*p);                       // *p is used as the anchor
```

这里出现了同样的问题。将p作为构造函数的参数时，它还没有值。但这不是错误，只是

一个警告。可以在使用指针之前将它初始化为空，从而避免这个警告。

```
Point *p = 0;                    // to avoid warning that pointer p has no value
p = new Point(*p,80,90);         // dynamically allocated Point object
Point p1(*p);                    // it is used as the anchor
```

将对象作为同一类的对象引用数据成员似乎是一个不错的主意，但由于其数据结构是递归的，因而会造成一些麻烦。除非必须要使用它，否则最好不要使用。

12.3.4 用静态数据成员作为其类自身的数据成员

可以将静态数据成员作为其类自身的数据成员，而且它比使用引用数据成员要简单得多。例如Point类的对象有一个原点，平面上所有的点都共用这个原点，因此可将这个原点表示为一个静态数据成员

```
class Point {
    int x, y;                                  // private coordinates
    static int count;
    static Point origin;                       // static object is ok
  public:
    Point (int a, int b)
      { x = a;  y = b;  count++; }  . . . } ;
```

初始化静态对象的语法与初始化内部数据类型的静态成员一样（详见第9章有关静态数据成员的讨论）。下面这行代码给出了静态对象的定义和初始化的例子。定义中的第一个Point表示正被定义的数据成员（这里是origin）的类型，第二个Point表示正被定义的数据成员属于Point类。在对象创建时，调用构造函数对该对象的字段进行初始化。定义中所指定的参数将作为构造函数的参数进行传递。参数的个数和类型决定了调用的构造函数。在这个例子中，调用的是带有两个参数的一般Point构造函数。

```
Point Point::origin(640,0);      // initialization using constructor
```

这与内部数据类型的静态数据成员定义类似。例如count，这个静态字段的类型是整数，类作用域操作符表明它属于Point类。该字段的初始值被设置为0。

```
int Point::count = 0;
```

与所有其他的静态对象类似，对象的创建与构造函数的调用不是那么清楚。如果程序中有几个静态变量，这些静态变量的创建次序并没有定义。即使在源代码中将它们按次序放好，也不能保证它们在创建和初始化时仍按照这个次序。惟一能保证的是所有这些静态变量都是在执行main()的第一条语句之前创建的。

但是对于Point类，这种保证是不够的。Point构造函数不仅被非静态的Point对象调用，而且要被先创建的静态对象origin调用。Point构造函数将使count的值加1。这就要求静态数据成员count要在origin对象创建之前创建。

静态数据成员的静态特性允许它可作为其类自身的成员函数的缺省参数。例如，可作为构造函数的缺省参数。非静态数据成员不能作为其类自身的成员函数的缺省参数。

```
class Point {
  int x,y;
  static int count;
  static Point origin;              // static object is ok
```

```
  Point &anchor;                    // reference or pointer is ok
public:
  Point (Point &focus = origin, int a=0, int b=0) : anchor(focus)
       { x = a;  y = b;  count++; }
  void set (int a=x, int b)         // error: non-static data member
  { x = a;  y = b; }
    . . . . } ;                     // the rest of class Point
```

静态数据成员在程序开始执行之前创建，即使没有创建类的对象，也可以访问这些静态数据成员。

在Point对象创建后，count和origin数据成员可被任意对象访问，访问的结果是相同的，因为这些数据成员是静态的。与非静态数据成员不同，访问静态数据成员时可以使用类名而不用目标对象名。

```
int main()
{ Point p1, p2(70,90);
  cout << "Number of points: " << p1.count << endl;     // prints 2
  cout << "Number of points: " << p2.count << endl;     // prints 2
  . . . }
```

与非静态数据成员不同，可使用带有作用域运算符的类名来访问静态数据成员，而不使用带有选择符的目标对象名。

```
int main()
{ Point p1, p2(70,90);
  cout << "Number of points: " << Point::count << endl;   // it also prints 2!
  . . . }
```

而且，在没有创建类对象时，下面的语法也是有效的。

```
int main()
{ cout << "Number of points " << Point::count << endl;     // it prints zero
  . . . }
```

程序12-3中Point类有两个静态数据成员：count和origin。即使它们是私有成员，也可以在类定义外对它们初始化。quantity()函数定义为静态的，可使用类作用域运算符访问（第一个调用），也可使用目标对象访问（第二个调用）。

程序12-3　使用静态数据成员和静态成员函数

```
#include <iostream>
using namespace std;


class Point {
    int x, y;                                   // private coordinates
    static int count;
    static Point origin;
  public:
    Point (int a=0, int b=0)                    // general constructor
      { x = a;  y = b;  count++;
        cout << " Created: x=" << x << " y=" << y
             << " count=" << count << endl; }
    static int quantity()                       // const is not allowed
           { return count; }
```

```
    void set (int a, int b)                    // modifier function
      { x = a;  y = b; }
    void get (int& a, int& b) const            // selector function
      { a = x;  b = y; }
    void move (int a, int b)                   // modifier function
      { x += a;  y += b; }
  ~Point()                                     // destructor
    { count--;
      cout <<" Point destroyed: x=" <<x <<" y=" <<y <<endl; }
} ;

int Point::count = 0;                          // initialization
Point Point::origin(640,0);                    // initialization

int main()
{ cout << " Number of points: " << Point::quantity() << endl;
  Point p1, p2(30), p3(50,70);                 // point of origin, point objects
  cout << " Number of points: " << p1.quantity() << endl;
  return 0;
  }
```

该程序的运行结果如图12-6所示。可以看到，即使没有显式地创建Point对象实例，count变量的值也不为0，这是因为静态对象origin的缘故。该对象在main()的第一条语句执行之前创建。当构造函数被调用时，创建对象的消息显示在最上面。程序终止后，静态数据成员才被撤销。因此看不到撤销这些静态对象的调试信息。

```
Created: x=640 y=0 count=1
Number of points: 1
Created: x=0 y=0 count=2
Created: x=30 y=0 count=3
Created: x=50 y=70 count=4
Number of points: 4
Point destroyed: x=50 y=70
Point destroyed: x=30 y=0
Point destroyed: x=0 y=0
```

图12-6 程序12-3的输出结果

我们交换一下定义count变量和origin变量的位置，程序的输出结果没有改变。

```
Point Point::origin(640,0);
int Point::count = 0;
```

这意味着编译程序能够识别这些变量之间的相互依赖关系，并确保在为静态对象origin执行Point的构造函数前，变量count的值是可用的。

本节讨论的是比较复杂的编程方法。要确信不是因为喜欢挑战才使用这些技术。我们需要替那些在未来几年中处理我们的代码的维护人员考虑。

12.4 容器类

我们可能不会每天都碰到上一节所讨论的有关复合类数据成员的特例。本节将讨论的有关复合类数据成员的特例则比较常用。即使不自己编写相似的类，也会用到别人编写的容器类。**容器类**（container class）是复合类的一个特例，当应用程序需要某种数据类型来容纳一

组动态的值时，容器类是很有用的。几乎所有的应用程序都需要一个（或多个）容器类。

虽然，我们在本章所讨论的第一个复合类的例子（Rectangle类）中包括了一些组件的集合，例如Point类型的对象，但这个集合不是动态的，与Rectangle对象相关联的Point对象的个数总是相同的：都是两个。实际上，如果客户代码中没有两个有效的Point对象，就不可能创建一个Rectangle对象。

应用程序经常需要一个容器类或集合类容纳一些数目可变的对象。通常，容器对象初始时为空，不包含任何组件。当应用程序执行时，组件对象将变得有效并添加到容器中暂时存储或进行处理。

例如，容器可以是客户缴费账户，组件可以是应该被处理的客户信用卡交易。这些处理可能包括计算总额与税收、打印报表或轮流访问容器中的每个组件等其他任务。此外，还有一些常规的任务，如测试给定的组件是否在容器中，从容器中删除一个组件，让容器变为可重用等。

绝大部分任务可用C++中进行数据表示的数组完成。数组是一种简单有效的数据类型，但它为客户代码提供的安全保障很不够：它不检查下标值的有效性，缺乏一些高级操作如添加一个组件、搜索一个组件等。这些操作必须由客户代码使用一些低级操作来完成，如给数组成员赋值，设置下一个成员的下标，检查下一个成员是否有效等。

容器类设计来为客户代码实现这些操作。客户代码可以要求容器在集合中增加一个组件，查找一个组件及访问集合中的每个组件等。容器对象完成这些操作，将客户代码与低级的处理细节隔离开来。这样，任务就由客户代码推向服务器代码（容器类）中。

本节我们讨论几个简单容器的例子，说明客户代码使用容器类的方法。

为简单起见，在这些例子中我们使用的Sample类的组件中只有一个数据成员，即double类型的value。Sample对象由某些外部处理产生：如证券交易所股票行情自动收报机用的纸带、病人监视设备、温度或压力观察仪等。在以下的例子中，我们从预先赋值的数组中提取值。

```
class Sample {                    // component class
  double value;                   // no pointers among data members
public:
  Sample (double x = 0)           // default/conversion constructors
    { value = x; }
  void set (double x)             // modifier
    { value = x; }
  double show () const            // selector
    { return value; } } ;
```

当某个类作为容器中的一个组件时，这个类必须要满足一定的支持容器对象的设计要求。最通用的需求是要具有以下性能：

- 缺省实例化。
- 可赋值性。

缺省实例化要求指的是，即使没有客户代码的任何输入数据时也能创建组件类型对象的能力。容器类可使用一个固定大小的数组表示组件对象。

```
Sample data[100];                 // container's data member
```

容器类另一个常用的设计是使用动态分配的组件数组。

```
Sample *data;                          // container's data member
data = new Sample[100];                // code in container constructor
```

不论是哪种情况，组件对象都是先分配而后赋值。为了满足该要求，组件类必须提供缺省的构造函数。否则，定义一个组件对象的数组会出现一个（或者一百个）语法错误。

这样，在组件类没有提供缺省的构造函数情况下，创建一个复合对象会产生语法错误。在容器的构造函数中使用成员初始化列表可消除这个错误。但这种方法只能在容器中的组件个数事先已知且不是很大时才能使用。成员初始化列表的本质局限性在于组件类的每个成员都必须出现在初始化列表中，而且组件名都必须显式地写出来。

与前面的例子类似，通过提供只有一个带缺省值的参数的转换构造函数，Sample类可以满足缺省实例化的要求。

可赋值性要求指的是，组件类型的对象能够被客户代码赋以新的状态的能力。由于容器的组件是先分配后赋值，它们应支持组件状态的改变。满足该要求的常用方法是为组件类提供赋值运算符函数。

```
data[i] = s;                           // code in a container method
```

该方法对于像Sample这样简单的类是可行的，因为它们没有动态的内存。但如果组件类中包含指针并动态管理堆内存，则需要一个重载的赋值运算符函数。

满足可赋值性的另一技巧是为组件类提供一个修饰符函数，该函数将改变组件对象的状态。

```
data[i].set(s);                        // code in a container method
```

Sample类提供了set()方法，允许不使用重载的赋值运算符函数而直接赋值，这样可以满足可赋值性的要求。

此外，人们还经常希望组件类满足另外两个要求，以支持：

- 拷贝实例化。
- 总排序语义（total order semantics）。

拷贝实例化要求指的是一个组件对象可被另一个组件对象初始化的能力。为组件类提供拷贝构造函数可以满足这个要求。如果容器类必须要返回一个组件对象的副本给客户代码，就必须提供拷贝构造函数。客户常常愿意使用组件对象的引用而不是整个副本。因此我们认为，在容器中使用拷贝构造函数的要求有些夸张。拷贝实例化很少出现在容器的算法中，支持拷贝实例化通常会鼓励按值传递对象参数或从函数中返回对象的值，这将带来一些不良后果。我们并不是在鼓吹不要提供拷贝构造函数。尤其当类动态管理堆内存时，不提供拷贝构造函数麻烦更大。如果不允许按值传递对象或返回对象值，我们推荐使用第11章结束处所提到的私有构造函数技术。

提示 不要急于为每个类提供拷贝构造函数。使用拷贝构造函数一方面会使程序速度变慢，另一方面会使程序更复杂。拷贝构造函数还会鼓励客户代码按值传递对象参数与按值返回对象。我们可以考虑定义拷贝构造函数为私有函数，这样可以在将来减少许多麻烦。

总排序语义指的是客户代码能在组件对象之间、以及组件和基本类型的值之间进行比较的能力。为组件类提供重载的比较运算符函数可以满足该要求。这是一个很有用的功能，尤

其当客户代码需要实现排序或搜索算法时。我们在例子中没有实现总排序语义，这是为了控制程序的大小，使程序易懂、易于管理。

在下面的例子中，我们将在History类的容器对象中保存Sample类的对象。History类将Sample对象保存在一个比较小的数组中（为简单起见，只有8个组件）。它允许客户代码在数组的给定位置设置Sample对象的值，将测量集合的值打印出来，并计算这些测量值的平均值。

程序12-4是容器类的第一个版本，程序的输出结果如图12-7所示。

程序12-4　有固定大小组件数组的容器类（数组溢出）

```
#include <iostream>
using namespace std;


class Sample {                          // component class
  double value;                         // sample value
public:
  Sample (double x = 0)                 // default and conversion constructor
    { value = x; }
  void set (double x)                   // modifier method
    { value = x; }
  double get () const                   // selector method
    { return value; } } ;

 class History {                        // container class
  enum { size = 8 };
  Sample data[size];                    // fixed-size array of samples
public:
  void set(double, int);                // modify a sample
  void print () const;                  // print history
  void average () const;                // print average
 } ;

void History::set(double s, int i)
{ data[i].set(s); }                     // or just:  data[i] = s;

void History::print () const            // print history
{ cout << "\n Measurement history:" << endl << endl;
  for (int i = 0;  i < size;  i++)                  // local index
    cout << "  " << data[i].get(); }

void History::average () const
{ cout << "\n Average value: ";                    // print average
  double sum = 0;                                  // local value
  for (int i = 0;  i < size;  i++)                 // local index
          sum += data[i].get();
  cout << sum/size << endl; }

int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;  // input data
  History h;                                         // default constructor
  for (int i=0; i < 9;  i++)                         // 8 slots are available
    h.set(a[i],i);                                   // set history
  h.print();                                         // print history
  h.average();                                       // compute average
```

```
return 0;
}
```

```
Measurement history:
 3  5  7  11  13  17  19  23
Average value: 12.25
```

图12-7 程序12-4的输出结果

注意容器类History的设计如何实现请求访问内存的算法。这意味着程序要在内存存储一些值供其他代码段以后使用。设计者依据这些合作的代码段所处位置的间距大小，以不同程度的耦合度来安排这些代码段（详见第8章中耦合度的讨论及相关的软件工程概念）。

一般而言，类设计者为了使类方法可以存储或检索某个值或某个变量，有以下选择：

- 全局变量或公有数据成员。
- 方法参数。
- 类数据成员。
- 方法中的局部变量。

当几个类必须共享信息而设计者又很难决定该信息属于哪个类时，可使用全局变量。这种类之间通信的方法具有最高程度的耦合度，应尽可能少地使用。当设计者选择某个类包含这些共享的信息时，可使用一个公有数据成员。由于其他几个类也需要该信息，因此该信息以公有数据成员的形式，使之对其他类有效。该方法的耦合度与使用全局变量的一样高，应少用。

当几个类使用全局变量或公有数据成员通信时，这应被视为出于设计的考虑。设计者应检查这些类之间的任务划分。使用全局变量或公有数据成员进行通信，容易让人产生疑虑：这种设计模式似乎将本来可以放在一起的处理步骤拆分开了，因此像这样“长距离”的通信需求可能会消失。

在深入讨论时，我们将集中在另外三种技术上。因为C++程序员每天都需要在这三种形式的耦合度中进行选择。

如果值或变量需要在类及其客户代码之间共享，应通过方法参数进行通信。例如，方法History::set()的参数被客户代码的main()函数和History类的成员函数set()所共享。

两个不同的类共享同一数据值时，这种方法是数据耦合度的最高形式。在两个不同的类中一致地处理该数据值要求这些类的设计者相互协作。如果这两个类都是由同一个人设计的，由于设计时间可能不同，这要求设计者要记住很多问题和限制。

只要可能就应该减少这种耦合度形式，而将所共享的数据值或变量的使用合并在一个类中．从而消除类之间的通信。但常常不这样做，因为面向对象程序的基础是一组相互协作的类而不是完全独立的类。因而，程序类之间的一些通信是合法的、有用的。但是，C++程序员应该常常考虑类之间通信的程度，并尽量消除过多的耦合度。

当变量需要在同一类的不同方法之间共享时，可通过类数据成员进行通信。例如，Sample类为每个类对象提供了存放数据成员value的存储区。到目前为止，这个设计策略可能看起来微不足道，但是请记住，正是这个设计支持了Sample的两个方法set()和get()

之间的通信。无论set()函数设置的值是什么（例如调用History类中的set()函数），这个值都会一直保存。当Sample类的客户代码以后调用get()函数时（例如在Histroy类的print()方法中或者average()方法中），get()函数获得的值与set()方法存储在该Sample对象中的值完全相同。

类似地，在History类的成员函数之间通信时使用了数据成员data[]。无论History的set()函数设置的值是什么，print()函数和average()函数都将从同一位置开始检索。这还可通过其他方式来实现，例如可将data[]数组作为main()中的局部变量，或作为文件中的全局变量并以参数形式传递给History方法。

下面的History类所处理的事情与程序12-4相同，但它不将Sample对象数组作为数据成员，而是从客户代码的main()中获取要操作的数据。

```
class History {
  enum { size = 8 };                                    // size of the data set
public:
  void set(Sample[], double, int) const;                // modify a sample
  void print (const Sample[]) const;                    // print history
  void average (const Sample[]) const;                 // print average
 } ;

void History::set(Sample data[], double s, int i) const
{ data[i].set(s); }                                     // or just:  data[i]=s;

void History::print (const Sample data[]) const         // print history
{ cout << "\n Measurement history:" << endl << endl;
  for (int i = 0;  i < size;  i++)                      // local index
    cout << "  " << data[i].get(); }                    // parameter data

void History::average (const Sample data[]) const
{ cout << "\n Average value: ";                         // print average
  double sum = 0;                                       // local value
  for (int i = 0;  i < size;  i++)                      // local index
       sum += data[i].get();                            // data from parameter
  cout << sum/size << endl; }
```

该设计虽然糟糕，但其语法正确、语义完整。主要缺陷是History类与其客户之间的频繁通信。在程序12-4中客户必须维护被History类维护的信息。

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;    // 9 values
  Sample data[9];                                 // whom should this data belong to?
  History h;                                      // default constructor
  for (int i=0; i < 9;  i++)                      // 8 slots are used
    h.set(data,a[i],i);                           // set history
  h.print(data);                                  // print history
  h.average(data);                                // compute average
  return 0; }
```

这种小的错误积累下来就会形成不同程序组件之间的大量的依赖性，从而破坏C++程序的质量。在设计程序时一定要时常考虑类之间的通信问题。

C++程序中最后一种通信方式是在类方法中使用局部变量，该方法最为友好。当成员函数需要存储一个数据值供以后调用此相同的函数时使用，就应该使用这种通信方式。例如，程

序12-4中的average()函数使用局部变量sum和i，来记录在某个特定的执行时刻被处理的数组成员和所积累的总和，所记录的sum和i将作为进一步计算数组成员总和的起始点。

与前面例子相似，这种通信方式可用别的方法实现。例如，考虑如下形式的History类，它提供了指定的数据成员记录数组成员及总和。

```
class History {                          // container class
  enum { size = 8 };
  Sample data[size];                     // fixed-size array of samples
  int i;                                 // index for method average()
  double sum;                            // tally for method average()
public:
  void set(double, int);                 // modify a sample
  void print () const;                   // print history
  void average () const;                 // print average
 } ;

void History::set(double s, int i)
{ data[i].set(s); }                      // or just:  data[i] = s;

void History::print ()                   // it modifies i
{ cout << "\n Measurement history:" << endl << endl;
   for (i = 0;  i < size;  i++)          // global, not local index
     cout << "  " << data[i].get(); }

void History::average ()                 // it modifies sum
{ cout << "\n Average value: ";          // print average
  sum = 0;                               // global value
  for (i = 0;  i < size;  i++)           // global index
                                         sum += data[i].get();
  cout << sum/size << endl; }
```

在这个版本的设计中，average()方法访问全局变量（数据成员）sum和i，而不是在方法执行过程中处理被分配的自动变量。这样当然会涉及性能上的问题。由于调用average()函数时不需要每次都分配空间，因此这样处理速度要快些。另一方面，不仅是为average()函数的调用过程给每个History对象分配空间、同时也是为这些对象的生存期而分配空间。该版本的average()函数编写起来更为容易——使用有效变量而不需要定义它们。在别的函数中也可以重用这些变量，例如，可以在print()函数中使用这个下标。

这种方法的主要涵义在于设计的质量。它比使用全局变量与其他函数通信要好一些。average()函数使用了全局变量（数据成员）sum和i与它自己进行通信（在重复下一次循环时），而不是与其他函数通信。但是，使用这种方法所设计的程序的质量仍不算很好，应避免使用。例如，可能因为别的目的而需要重用全局变量（与我们在print()函数中重用下标以避免额外的声明相类似），这种重用将可能导致冲突。C++所支持的软件工程思想是：应让每个函数定义它自己单独的局部变量，并合理地、不冒冲突风险地使用这些变量。

要尽量使用最低的耦合度。如果在成员函数中用局部变量就能够处理的工作，则不要将这些局部变量提升为类的数据成员。如果同一类的几个成员函数都要访问相同数据，则将这些数据作为类的数据成员，不要在类的客户代码中作为参数传递它们。如果成员函数需要另一个类中定义的数据，则以参数形式传递该数据，而不要将它作为全局变量或另一个类的公有数据成员。

提示　对于C++中不同程序段之间的通信，要使用最低程度的耦合度：通过方法中的局部变量通信。如果这样不能满足数据流的要求，则使用类数据成员。只有当这样做还不够时，才以方法参数的形式传递信息。总而言之，要避免使用全局变量。

让我们将这些软件工程基本原则应用到程序12-4的类设计中。History类是一个非常简单的容器类，不能为客户代码提供任何保障，如防止容器溢出或引用数组中并不存在的数据项。在这个版本的容器中，只有8个槽存放Sample对象。尽管有此限制，main()中的客户代码仍将9个数据值放到容器中。编译程序当然不能置之不理。尽管程序是不正确的，但是操作系统运行该程序时没有出现可见的错误（如图12-7所示）。这是使用容器的应用程序中常出现的问题。客户代码和容器类之间的任务划分可能不同，但必须实现防止容器溢出，而且这应是容器类的任务，而不是客户代码的任务。

当一个新的Sample值插入到容器时，程序12-4中的客户代码不仅要给定插入的值，还要给定插入值的下标。但该方法与软件工程基本原则相违背——应将任务推给服务器类，即History容器。这对于像该例子这样简单的算法而言没有什么影响（所有的输入值都是一次性输入的，没有其他操作干涉该容器对象），但客户代码有其他更重要的任务，不应该关注容器还有多少空余空间。监测容器状态应是容器对象本身的任务。

按刚刚讨论过的类通信基本指南的观点，客户代码和History::set()之间界面的耦合度太高了。客户代码被迫将有关下标的信息作为一个额外的参数来传递。根据类通信基本原则，耦合度较低的一个办法是通过类数据成员通信。为了改善设计方案，我们将下一个受影响的Sample对象的下标的相关信息保存在History类中，而不是客户代码中。将本属于一个整体的程序拆开会导致程序员之间额外的交流及函数之间额外的耦合度。

程序12-5是一个更好的容器设计。带有两个参数的History::set()方法被只有一个参数的History::add()方法取代，方法的参数被插入到容器的末尾。容器新增了一个数据成员，即下标idx，它将监视容器对内存的使用。客户代码并不知道容器是否已经满了，它只是传递要加到add()方法中的数据值。

既然现在客户代码不控制容器内存的使用，那么就由容器自己负责记录已使用的和仍可用的内存，并负责控制溢出。因此，容器知道其内存结构及其局限。在程序12-4中，客户代码决定下一个数据值要去的地方，而不需要初始化容器对象。在程序12-5中，由容器决定下一个数据值要去的地方，容器必须被初始化为空，以确保第一个到达的数据值存放到第一个槽中。因此，这里的History类有一个缺省构造函数。在该构造函数中，History类将下标idx设置为0。在add()方法中，容器类检查数组是否已满：如果有空余空间，add()方法使用另一个空余的槽，使下标idx加1以指向下一个空余的槽。如果没有空余的槽给传递过来的数据，add()方法将忽略客户要求，什么也不做。

当然，一个比较好的做法是，将Sample添加到History是否成功的消息告诉客户代码。这样允许客户代码启动一些恢复措施或将某些情况通知程序用户。但这样比较浪费程序员精力。我们应该避免这样，不是因为反馈信息给调用者不重要，而是因为内存溢出是不可容忍的。所有固定大小的数组都应该只用来开发快速原型。在对算法调试后，这些数组应被动态数组所代替，所采用的方法与第6章所介绍的类似。（当然，除非开发的是一个实时系统，那就是另外一个问题了）。

程序12-5的输出结果与程序12-4的相同。

程序12-5 有固定大小的组件数组和溢出控制的容器类

```
#include <iostream>
using namespace std;

class Sample {                          // component class
  double value;                         // sample value
public:
  Sample (double x = 0)                 // default and conversion constructor
    { value = x; }
  void set (double x)                   // modifier method
    { value = x; }
 double get () const                    // selector method
    { return value; } } ;

class History {                         // container class: set value
  enum { size = 8 };
  Sample data[size];                    // fixed-size array of samples
  int idx;                              // index of current sample
public:
  History() : idx(0) { }                // make array empty initially
  void add(double);                     // add a sample at the end
  void print () const;                  // print history
  void average () const;                // print average
 } ;

void History::add(double s)
{ if (idx < size)
    data[idx++].set(s); }               // or just:  data[idx++] = s;

void History::print () const
{ cout << "\n Measurement history:" << endl << endl;
  for (int i = 0;  i < size;  i++)      // local index
    cout << "  " << data[i].get(); }

void History::average () const
{ cout << "\n Average value: ";
  double sum = 0;                       // local tally
  for (int i = 0;  i < size;  i++)      // local index
    sum += data[i].get();
  cout << sum/size << endl; }
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;    // input data
  History h;                                           // default constructor
  for (int i=0; i < 9;  i++)                           // it is protected from overflow
    h.add(a[i]);                                       // add history
  h.print();                                           // print history
  h.average();                                         // print average
  return 0;
  }
```

注意最小可见度原则：容器类History要尽可能少地向其客户展露内部结构及内存限制。

程序12-4和程序12-5中的容器类的一个重要局限性是它们必须在客户访问容器中的组件之前被装满数据。容器方法print()和average()方法遍历容器数组直到数组末尾。另一个重要局限性是从客户代码的角度而言，组件上的所有操作都是单个操作，客户代码经常要单个地访问组件，再对每个组件恰当地执行或者跳过操作。

第一个不足可通过在容器类中增加另一个数据成员count来弥补。

```
class History {                          // container class: set value
  enum { size = 8 };
  Sample data[size];                     // fixed-size array of samples
  int count;                             // number of valid elements
  int idx;                               // index of the current sample
public:
  History() : count(0), idx(0) { }       // make array empty
  void add(double);                      // add a sample at the end
  . . . } ;                              // rest of History class
```

数据成员count在构造函数中（在成员初始化列表中）被设置为0，每当增加一个新的组件添加到容器中时它将被更新。

```
void History::add(double s)
{ if (count < size                 // check for available space
    data[count++].set(s); }        // use next space, update count
```

即使容器未满，容器成员函数也可使用count来处理正确数目的组件。因此，average()函数可以使用count限制用来计算的组件个数。

```
void History::average () const
{ cout << "\n Average value: ";
  double sum = 0;
  for (int i = 0;  i < count;  i++)
       sum += data[i].get();
  cout << sum/count << endl; }
```

第二个不足可通过提供迭代方法来弥补。该方法允许客户代码访问容器中的每个组件，处理各种事情。迭代方法的形式多样。

对于组件上的迭代操作，我们使用已有的数据成员idx，在迭代操作开始时设置idx为0，迭代每执行一次，idx加1。我们在容器类中添加了方法getFirst()让客户代码开始迭代操作。

```
void getFirst()
  { idx = 0; }                         // set to start of data set
```

我们还在容器类中增加了方法getNext()让客户代码进行下一步迭代操作。

```
void getNext()
  { ++idx; }                           // move to next element in set
```

为了让客户代码访问容器中的当前组件，我们增加了getComponent()方法：

```
Sample& getComponent()
  { return data[idx]; }                // get the reference, not value
```

注意在这里我们没有返回当前对象，而是返回当前对象的引用。因此我们不必担心组件类是否有拷贝构造函数。如果组件类有拷贝构造函数，我们也不必担心拷贝组件会花费太长时间。我们回避了整个问题。

为了停止重复操作，我们必须提供一个方法，该方法在可继续迭代操作时返回真，当容器中不再有可迭代的元素时返回假。

```
bool atEnd()
  { return idx < count; }          // true if there are more elements
```

这样，客户代码中的迭代循环将是如下形式：

```
for (h.getFirst();  h.atEnd();  h.getNext())      // go until end
   cout << "  " << h.getComponent().get();        // print components
```

容器设计者常常将getNext()函数和atEnd()函数合并为一个函数，该函数将下标值加1，如果还有可迭代的元素时返回真。

```
bool getNext()
{ return ++idx < count; }      // move to next element in set
```

程序12-6的容器类中实现了重复方法。我们去掉了容器方法print()，让客户代码负责驱动迭代操作并访问组件元素的状态。于是有些任务就从容器类中拉到客户代码中。这样处理不是很好，但这是在容器类中增加了迭代操作的自然结果。

程序12-6的输出结果与程序12-4、程序12-5的相同。

程序12-6 带有固定大小的组件数组及迭代操作的容器类

```
#include <iostream>
using namespace std;

class Sample {                          // component class
  double value;                         // sample value
public:
  Sample (double x = 0)                 // default and conversion constructor
    { value = x; }
  void set (double x)                   // modifier method
    { value = x; }
 double get () const                    // selector method
    { return value; } } ;

 class History {                        // container class: set value
  enum { size = 8 };
  Sample data[size];                    // fixed-size array of samples
  int count;                            // number of valid elements
  int idx;                              // index of the current sample
public:
  History() : count(0), idx(0) { }      // make array empty
  void add(double);                     // add a sample at the end
  Sample& getComponent()                // return reference to Sample
  { return data[idx]; }                 // can be a message target
  void getFirst()
  { idx = 0; }                          // set to start of data set
  bool getNext()
  { return ++idx < count; }             // move to next element in set
  void average () const;                // print average
} ;
void History::add(double s)
{ if (count < size)
    data[count++].set(s); }                          // or just:  data[i++] = s;

void History::average () const
{ cout << "\n Average value: ";
  double sum = 0;
  for (int i = 0;  i < count;  i++)
          sum += data[i].get();
  cout << sum/count << endl; }
```

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;          // input data
  History h;                                                 // default constructor
  for (int i=0; i < 9;  i++)
    h.add(a[i]);                                             // add history
  cout << "\n Measurement history:" << endl << endl;
  h.getFirst();                                              // work is pushed up
  do {
     cout << "  " << h.getComponent().get();                  // print components
    } while (h.getNext());
  h.average();
  return 0;
  }
```

有些C++程序员喜欢将迭代方法绑定到一个单独的迭代器类中，并将该迭代器类与容器类链接起来。需要指出：这样做会增加程序的复杂性。我们不这样处理，而是想办法消除该容器类中最重要的局限性：即容器所能容纳的组件个数的限制。在前面的几个版本的容器中，它们的容量在创建容器时就固定下来。如果客户代码想将超过容器容量的更多的元素放到容器中，情况就会很糟糕，而且容器类也没有任何解决办法。

实际上，消除该局限不是太难。容器类应该处理的工作只是分配新空间，将已有的数据拷贝到新空间，删除已有的空间，使用新分配的空间直到用完。分配新空间的一个有效策略是将数组的大小增加一倍。

```
void History::add(double s)
{ if (count == size)
  { size = size * 2;                                         // double size if out of space
      Sample *p = new Sample[size];
    if (p == NULL)
      { cout << " Out of memory\n";  exit(1); }              // test for success
    for (int i=0;  i < count;  i++)
      p[i] = data[i];                                        // copy existing elements
    delete [ ] data;                                         // delete existing array
  data = p;                                                  // replace it with new array
  cout << " new size: " << size << endl; }                   // debugging
 data[count++].set(s); }                                     // use next space available
```

为了让该算法正常工作，数据成员data应是指向动态分配的Sample对象数组的指针。这要在构造函数中做些修改。

```
class History {                                              // container class: set value
  int size, count, idx;
  Sample *data;                                              // dynamic memory
 public:
  History() : size(3), count(0), idx(0)                      // make array empty
  { data = new Sample[size];                                 // allocate new space
    if (data == NULL)
       { cout << " Out of memory\n";  exit(1); } }
   . . .} ;                                                  // the rest of class History
```

程序12-7是这种版本的容器类。为简单起见，我们将容器的初始大小设为一个非常小的值（只有3个组件）来演示该算法的工作机制。该程序的输出结果如图12-8所示。在屏幕的最上方，调试信息表明容器的容量从3增加到6（当第4个值插入到容器时），然后又从6增加到12（当第7个值插入到容器时）。

程序12-7 动态分配内存的容器类

```
#include <iostream>
using namespace std;

#include <iostream>                    // dynamic container of variable size
using namespace std;

class Sample {                         // component class
  double value;                        // sample value
public:
  Sample (double x = 0)                // default and conversion constructor
    { value = x; }
  void set (double x)                  // modifier method
    { value = x; }
 double get () const                   // selector method
    { return value; } } ;

 class History {                       // container class: set value
  int size, count, idx;
  Sample *data;
 public:
  History() : size(3), count(0), idx(0)          // make array empty
  { data = new Sample[size];                     // allocate new space
    if (data == NULL)
      { cout << " Out of memory\n";  exit(1); } }
  void add(double);                              // add a sample at the end
  Sample& getComponent()                         // return reference to Sample
  { return data[idx]; }                          // can be a message target
  void getFirst()
  { idx = 0; }
  bool getNext()
  { return ++idx < count; }
  void average () const;                         // print average
  ~History() { delete [ ] data; }                // return dynamic memory
} ;

void History::add(double s)
{ if (count == size)
  { size = size * 2;                             // double size if out of space
      Sample *p = new Sample[size];
    if (p == NULL)
      { cout << " Out of memory\n";  exit(1); }  // test for success
    for (int i=0;  i < count;  i++)
      p[i] = data[i];                            // copy existing elements
    delete [ ] data;                             // delete existing array
    data = p;                                    // replace it with new array
    cout << " new size: " << size << endl; }     // debugging print
   data[count++].set(s); }                       // use next space available

void History::average () const
{ cout << "\n Average value: ";
  double sum = 0;
  for (int i = 0;  i < count;  i++)
      sum += data[i].get();
  cout << sum/count << endl; }

int main()
```

```
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;     // input data
  History h;
  for (int i=0; i < 9;  i++)
    h.add(a[i]);                                        // add history
  cout << "\n Measurement history:" << endl << endl;
  h.getFirst();                                         // work is pushed up
  do {
     cout << "   " << h.getComponent().get();           // print each component
    } while (h.getNext());
  h.average();
  return 0;
  }
```

```
new size: 6
new size: 12

Measurement history:

 3  5  7  11  13  17  19  23  29
Average value: 14.1111
```

图12-8　程序12-7的输出结果

注意，动态内存管理需要在撤销容器对象时使用析构函数将动态内存还回给堆。

该程序也用到了一些比较复杂的设计：组件可以进行排序、查找、删除、插入、修改及比较。设计容器类是件很有趣的事，使用容器类也同样有趣。标准模板库（Standard Template Library）中有大量的容器类供使用。这里的讨论可以作为对这个库使用的概念的很好介绍。

12.4.1　嵌套类

让我们回过头来研究程序12-5。注意在这里，我们不用添加迭代函数，客户代码也不使用Sample对象。

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;     // input data
  History h;                                          // default constructor
  for (int i=0; i < 9;  i++)                          // it is protected from overflow
    h.add(a[i]);                                      // add history
  h.print();                                          // print history
  h.average();                                        // print average
  return 0; }
```

实际上，是History对象访问Sample类。C++允许程序员在客户类中定义服务器类，这样嵌套的类名在聚集类之外是不可见的。

```
class History {
  class Sample {                        // not visible outside of the client scope
    double value;                       // private data: it could be public here
  public:
    Sample (double x = 0)
     { value = x; }
    void set (double x) { value = x; }
    double show () const { return value; }
     } ;                                // end of the nested class definition
```

```
    int size, count, idx;
    Sample *data;
public:
  History() : size(3), count(0), idx(0)   // make array empty
  { data = new Sample[size];          // allocate new space
    if (data == NULL)
       { cout << " Out of memory\n";  exit(1); } }
    . . .} ;                          // the rest of class History
```

嵌套类既可在客户类的私有部分定义，也可在其公有部分定义。无论哪种情况，嵌套类对程序的其他部分是隐藏的，只有在定义嵌套类的复合类作用域（括号）中才能使用嵌套的类名。

在复合类之外的其他作用域中不能使用嵌套类来声明变量。隐藏类定义与隐藏数据成员的方式相同。因此，如果必须在复合类之外用到嵌套的类名，则要使用作用域运算符。

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;     // input data
  History h;                              // default constructor
  for (int i=0; i < 9;  i++)              // it is protected from overflow
    h.add(a[i]);                          // add history
    h.print();                            // print history
    h.average();                          // print average
  Sample s = 5;                           // not ok for a nested class
  History::Sample s = 5;                  // scope operator resolves the problem
  return 0; }
```

要使最后的语句合法，Sample类必须定义在其客户类History的公共部分。这样，History类的客户代码可使用带有聚集类名作用域的Sample类名。

C++允许在同一语句中将类的定义与类实例的定义结合起来。但是，这不是很好的办法，因为类实例常常是全局的。

```
class Sample {                        // global in a file
   double value;
 public:
   Sample (double x = 0)
    { value = x; }
   void set (double x)
    { value = x; }
   double show () const { return value; }
    } s1,s2;                          // global objects of class Sample
```

但这对于嵌套类是比较合适的，因为数据成员通常在类作用域内是全局的。

```
class History {
  class Sample {        // not visible outside of the client scope
    double value;
  public:
    Sample (double x = 0)
      { value = x; }
    void set (double x)
      { value = x; }
    double show () const
      { return value; }
     } *data;                    // defining data members at the end
  int size, count, idx;
public:
```

```
History() : size(3), count(0), idx(0)    // make array empty
{ data = new Sample[size];                // allocate new space
  if (data == NULL)
    { cout << " Out of memory\n";  exit(1); } }
. . .} ;                          // the rest of class History
```

不要过多地使用该方法，它会使程序模糊难懂。

如果其他的类需要将该组件类作为它们的服务器类，则不能将该组件类定义为嵌套类。如果其他的类不需要组件类，嵌套类没有其他的作用了。但若程序的几个部分因为完全不同的目的而使用相同的类名时，使用嵌套类可以避免名字冲突。嵌套的类名不会使程序的名字空间混乱。

例如，某些类想使用Sample定义其他的度量结果，用不同的传感器，不同类型的数据值，甚至数据值的个数也不同。上面的Sample类可能不能满足这些要求，因此程序的那个部分可能需要使用它自己的Sample类。为了解决冲突，可使用两个不同的类名。例如：Sample1和Sample2。但是只使用一个名字并使它成为嵌套类会更加方便。

像Node类这样的类名也常用来表示链表的组件。如果同一Node类对于另一个链表结构（如栈或二叉树）也很有用，则应在全局名字空间声明该Node类。由于不同的链接结构常常包含不同的信息项，同一Node类不能为所有的链接结构服务。

这种情况下，在每个容器类中将Node类定义为局部类可能有更大的优势。这将消除潜在的名字冲突。消除全局名字可以降低开发不同容器类的编程小组之间的协调强度。

```
struct Node {           // a good candidate to be a nested class
  char* value;          // pointer to information contents (e.g., a word)
  Node* next; } ;       // pointer to next node
```

12.4.2　友元类

类的设计者应让客户代码比较方便地访问类，而不用创建过多的不必要的数据依赖。

很少将数据成员定义为类的公共成员，因为这样做没有显式地指定访问数据的限制，因此可能公开了太多的访问权限。一旦调试出现了问题，就不知道应去检查哪些客户；另一方面，如果改变了数据表示形式，也不清楚哪些客户会受其影响。

因此，大部分非成员函数（全局函数或其他类的成员函数）只能访问类的公共部分。而对于私有数据与私有函数，只有通过公共成员函数来访问。

通过类的成员函数访问非公共类成员会增加客户代码的任务。正如在第11章中介绍的，C++提供了一种扩展的访问类私有成员的机制：如果将一个非成员函数声明为类的友元，则该友元函数拥有与类的成员函数相同的访问权限。

但友元函数的确破坏了数据封装性，应少使用。另一方面，友元列表是类定义中的一个显式部分，在需要时可用它检查和标识受影响的客户。由于友元函数可以访问类对象的私有数据，在与类无关的情况下使用它是没有价值的。

友元并不只限于独立的全局函数。友元函数也可以是另一个类的成员。一个类的成员函数也可作为另一个类的友元。

可以将一个类的所有函数定义为另一个类的友元。这样该类的成员函数可以不使用访问函数而直接访问另一个类的私有成员。

也可以只指定某些客户成员函数作为服务器类的友元（需要事先声明）。

当某个类只为一个客户类服务时，如果将客户类作为服务器类的友元，就可以简化客户类和服务器类，这样客户类（如History）的每个成员函数都能访问服务器类（如Sample）的非公有成员。相关语法是，在类名前要使用friend关键字（不论在服务器类的哪个部分）。

```
class Sample {
          friend History;           // friend declaration
    double value;
  public:
     Sample (double x = 0)
       { value = x; }
     void set (double x)
       { value = x; }
     double show () const
       { return value; }
      } ;
```

除非已经定义History类，否则上面的定义将是一个语法错误。但是，History类又不能在Sample类之前定义，因为History类中使用了Sample类名。

```
class History {
   int size, count, idx;
   Sample *data;                          // circular dependency
 public:
  History() : size(3), count(0), idx(0)   // make array empty
  { data = new Sample[size];              // allocate new space
    if (data == NULL)
        { cout << " Out of memory\n";  exit(1); } }
   . . .} ;                               // the rest of class History
```

这是一个典型的代码循环依赖——一方面，History类使用了Sample名，Sample类要在History类之前定义。另一方面，Sample类使用了History类名，History类也应在Sample类之前定义。

有两种方法告诉编译程序Sample类定义中History的意义。一个方法是使用事先声明指定一个名字为类名。这里，History名被定义为类名而没有给出其定义细节。

```
class History;                      // class is declared elsewhere

class Sample {
       friend History;              // friend declaration
    double value;
  public:
    Sample (double x = 0)
      { value = x; }
    void set (double x)
      { value = x; }
    double show () const
      { return value; }
     } ;
```

另外一种方法是在友元声明中直接定义History是一个类。

```
class Sample {
       friend class History;        // friend declaration
    double value;
  public:
```

```
    Sample (double x = 0)
      { value = x; }
    void set (double x)
      { value = x; }
    double show () const
      { return value; }
     } ;
```

如果只有一种方法解决问题可能会比较好，但如果链接程序能解决这些交叉引用就更好了。

现在，History类的成员函数可以直接访问Sample类的非公共数据。

```
void History::print () const
{ for (int i = 0;  i < count;  i++)          // print valid elements only
//      cout << " " << data[i].show();
       cout << " " << data[i].value;         // no need to use methods
    cout << endl; }
```

而且，Sample类也不用再提供访问成员函数了，因为友元History类不再需要这些函数。

```
class Sample {
       friend class History;      // friend declaration
     double value;
  public:
     Sample (double x = 0)
       { value = x; }             // no need for other member functions
       } ;
```

软件工程和程序设计方法学对友元持怀疑态度，因为友元可能破坏了信息隐藏。友元的确使得程序设计更复杂难懂。

也可以不使用友元，而将Sample类作为History类的嵌套类，并将其字段定义为公共的。由于只有History类访问这些字段，因而保持了信息隐藏。

```
class History {
  intsize, count, idx;
struct Sample {                              // not visible outside of the client
scope
    double value;                            // data member IS public here
  Sample (double x = 0)
    { value = x; } } *data;                  // dynamic History data
public:
  History() : size(3), count(0), idx(0)      // make array empty
  { data = new Sample[size];                 // allocate new space
    if (data == NULL)
       { cout << " Out of memory\n";  exit(1); } }
   . . .} ;                                  // the rest of class History
```

这里的History类的成员函数的实现和前面一样，也拥有访问Sample类的非公有成员的所有权限。

要谨慎使用友元，考虑选择其他方法。

12.5 小结

本章我们讨论了将C++类作为一个类关系的组件，即类复合。这样我们在考虑类时，不再

认为它们只是一段段独立的代码，而是相互关联、相互协作的组件。

聚集是类之间最常用的关系。将类对象作为其他类的数据成员时涉及到一系列的技术与概念问题：如何定义类组件，如何适当地初始化这些组件以使组件能被其客户（即组件类）使用等。

我们也讨论了使用指针和引用链接类对象的其他方法。这是一种更加有效的建立对象关系的方法，应用十分广泛。详细讨论这些编程技术已经超过了C++语法的学习内容。但是在我们的职业编程生涯中，我们可能一直需要把对象放在别的对象中，也可能一直需要使用指针和引用连接对象。

我们还讨论了类聚集的一些特殊情况，如容纳一些组件对象的容器类。这也是类之间常见的关系，它可以由多种方式来实现。同样地，无论如何，我们都经常要创建容器类或使用库容器或者两者兼需。

享受使用这些技术的乐趣吧。

第13章　如何处理相似类

同前面的几章一样，这一章介绍更多有关C++的语法：关键字、冒号、初始化列表等。但大家的注意力千万不要被这些语法细节所分散。

在这本书的第一部分，我们已经学习了C++计算方面的知识。也了解了传统的编程知识，如数据类型、标识符、关键字、表达式、一般语句、条件语句、循环和其他的控制结构。不仅是在C++中，而且在任何编程语言中，使用这些机制的技术都是做其他工作的必要前提。程序员可使用这些技术编写代码以完成其编程目标，并生成需要的结果，这些结果是以可计算性的需求表达的。

在这本书的第一部分，我们也学习了一些关于聚集的方法：把一些数据元素集中放到数组、结构或者其他程序员定义的数据类型中，也可以把一些语句和控制结构集中放到函数中。用C++做这些工作比用其他语言更复杂，尤其当它涉及到处理名字作用域、传递参数、返回值、指针和引用时。我们也体会到了C++动态内存管理所带来的快乐和风险。这些机制直接用来将程序划分成互相协作的部分，但是，它们的直接目的更多是为了程序员的方便而不是为了达到程序的计算目的。可以有多种不同的设计方案实现计算目的，但是程序的质量（从维护性角度来看）可能有很大不同。

将C++代码元素正确地组合起来，以及将不应该在一起的代码元素分开，这些技能是编写可维护、可修改的C++程序的必要前提条件。

在本书的第二部分，我们学习了怎样应用在第一部分所学的知识编写C++类。前面已讨论过类的语法、类的作用域、数据成员、成员函数、对数据和函数的访问、具有语法及含义的消息、对象初始化、不同类型的构造函数与析构函数、静态数据和函数。我们也了解了运算符函数，它使C++代码更好，但同时也使类的设计更复杂。还学习了友元的概念。我们了解怎样去辨识类设计中的有害成分，以及怎样避免它们对程序造成的负面影响。在编程中使用类将使C++比其他语言复杂得多，但值得这样做。

将相关的数据和函数放到一起（放在同一类中）是面向对象编程的必要前提条件。传统编程和面向对象编程主要的不同是：传统的C++程序是建立在多个协作的全局函数上，这些函数将每个操作都绑定在一起；与此相对，面向对象程序是建立在多个相互协作的类之上的，这些类绑定了数据和基于数据的操作。

然而，这本书的前两部分仅仅是面向对象编程的基础。在所有的例子中，我们只处理了一个类，因为关心的是类设计的细节，而不是类之间的关系。在本书的第三部分，我们开始学习将C++程序构造为一组相互协作的类的集合。这需要在C++程序中实现类之间的关系。在第12章中，我们已经看到怎样将一个对象（服务器对象）用作另一个对象（它的客户对象）的元素。元素对象的成员函数为复合类的成员函数服务。这是最常见的对象之间的简单关系。

一个对象的数据成员的指针可以指向另一个对象。客户对象可以通过将消息发送给指针数据成员，以访问服务器对象的成员函数。一个对象也可以作为客户对象的引用数据成员。从语法上来看，这与简单类的复合是类似的，但确切地说，两种情况下的对象之间的关系是

十分不同的。

通过简单的类复合，服务器对象（一个元素）变成一个客户对象（一个复合对象）的数据成员。在这种关系下，客户对象独占这个数据成员，并使用它的元素对象。当服务器对象是客户对象的一个引用（或一个指针）数据成员时，服务器对象可以被几个客户对象共享；几个对象可以指向同一个服务器对象。服务器对象的改变将影响到客户对象（或几个客户对象）的状态。讨论一般情况下究竟是独占还是共享元素会更好是毫无意义的。但在大多数实际情况下，一个关系比另一个关系从某种意义上而言“更好”，是因为它能够更好地体现C++程序所建模的现实生活中的实体之间的关系。重要的是要选择最适合给现实生活建模的对象关系。

我们也看到一个在对象之间非常普遍的关系，即一个对象实现为一个容器，并且有一组（而不是单个）其他类的对象作为这个容器的元素。这种关系在C++程序中是很常见的。

本章，我们将继续学习C++代码各部分之间的协同合作关系，也将介绍在C++中如何通过继承机制反映一个应用程序中各个类之间的关系。目前大家可能还不清楚对象之间相互关系和类之间相互关系的不同，学习完本章后就会明白了。

在C++程序中会经常用到继承。这个强有力的机制有利于重用C++设计，有利于在程序员之间划分工作，也有利于在C++程序中引入模块化。为了能正确地使用继承，首先应学习其语法、派生对象实例化的方法、访问元素的技术、函数调用规则等等。而且，为了更好地完成任务还要学习如何恰当地选择继承或者类复合。由于继承是强有力的机制，C++程序员有时会过分地使用继承，结果导致产生了额外的关系和依赖性，使得程序难以理解。

13.1 相似类的处理

我们的程序根据现实生活中各种各样之对象的数据（对象状态）和操作（对象行为）进行建模。在面向对象设计的大前提下，每个设计人员都可以决定每个类中应该包括哪些内容。理论上，对现实生活中的实体进行建模应该能反映实际对象之间的“共同特征”，比如反映库存清单、事件记录、或银行账户之间的共同特征。

当然，这些“共同特征”是可观察到的，C++有不同的机制来表示实体之间不同程度的相似性。

C++提供的第一种提取现实生活中对象之间共同特征的机制就是构造类，当我们确信这些对象都具有一些共同的属性和共同的行为模式等特征时，我们就使用构造类去提取对象之间的共性。这些对象的不同表现在状态属性的值上，例如：不同矩形的顶点有不同的坐标值，不同的库存项目有不同的标题，每个账户也有它自己的余额和账户名。相同的因素是每个长方形都有顶点，每个项目都有标题，每个账户都有余额和账户名。假如一个账户需要指定一个利率而另外一个账户不需要，正常情况下，这两个账户不应被看作同一类中的对象。

通常，情况并不一目了然。比如，在具体应用中，一个仓库里的每个螺丝钉对象具有区别于其他螺丝钉的特征。我们需要为每个螺丝钉对象设计一个单独的类，给每个类一组数据和一些成员函数去描述每个螺丝钉，并为每个类取一个惟一的类名。这些名字反映了每个螺丝钉在该应用中的惟一特征，例如，`RustyBolt`、`UglyBolt`和`BoltFoundInPothole`等。但这样做会使事情变得复杂，只有在不同的螺丝钉之间没有共同特征并且行为不同时才有意义。

然而，假如仓库中所有的螺丝钉都有大量的共同之处，那么可以为应用程序中所有的螺丝钉的数据成员取相同的名称（事实上，将这些共同之处抽取之后，构成类的属性），这样可避免一定要用不同类的对象代表不同的螺丝钉。我们可以只用一个类，例如Bolt，然后用此类的对象代表应用程序的每一个螺丝钉。这个类的属性有购买日期，供货商名称和螺距等。与此类似，如果同一组属性（颜色、材质、尺寸等）足够描述每一个螺母对象，也可以用Nut类的对象来代表仓库中所有的螺母。

如果Bolt类、Nut类和其他仓库项目的数据成员都采用相同的名称来标识，可以放弃螺母和螺丝钉的区别，进而使用一个InventoryItem类代表这些（螺母、螺丝钉）不同的对象。如果从应用的角度来看所有的螺丝钉都是相同的，就可以用单个对象来代表所有的螺丝钉，并且在类的属性中定义螺丝钉的数量。既然所有的螺丝钉都是相同的，那么在其他螺距方面的差别也就不重要了，但如果其他螺距方面的属性很重要，则不能采用这种类的设计方案。

如果应用程序只是对螺母和螺丝钉及其他的一些库存项目的总价值感兴趣，那么我们就可以用一个Asset类来代表库存信息，令其属性满足应用程序的需要。

然而，通常共性可以存在于若干个类中：这组对象有基本的相似之处，但仍有一些不同的属性和操作。

例如，小螺丝钉以每100个为单位来衡量其重量；而大螺丝钉则以每个为单位来衡量其重量，并且还可能有一属性表示大螺丝钉可以承受的最大压力。

类似地，按小时计酬的合同工可能将一星期内工作的小时数指定为数据成员。而正式工可能拥有同样的属性（名字、地址、雇佣日期等），但是需要指定每年的薪水，而不是每小时的薪金和工作的小时数。

有一些对象组可能有一些不同的操作集或者提供了附加的操作，例如，存钱会有利息，而用支票账户却要支付处理费。简单地把所有的这些特征合并在一起，虽然可以适应客户代码的需求，但这样做本质上是不安全的。客户代码可能误认为某一特定对象拥有其他对象的一些特点，从而不正确地使用了对象。例如，客户代码可能会在用支票账户时付给利息而在存钱时却要求付处理费。

将所有的属性和操作合并到一个类以满足所有的情况，这仍是一种可行的抽象方法，但这就需要客户代码根据对象内在的特征去确定对象的使用。

13.1.1　把子类的特征合并到一个类中

考虑一个大家都很熟悉的例子，这个例子的应用背景大家或者亲身体验过或者听别人谈论过。

我们讨论一个简单的Account（账户）类，它有一个数据成员balance以及成员函数withdraw()和deposit()。执行一个支票账户的取款操作时，该账户将被扣去一次处理费（如20美分）。执行一个储蓄存款账户的存款操作时，添加日利息（如以年利率6%计算）。这些处理费和利率都被表示为Account类的数据成员。为了简化例子，我们不讨论指定和修改数字变量值的技术，也不讨论其他无数的实际细节，例如账户持有者姓名、地址、年龄、社会保险号、透支费，以及其他和银行业务相关的细节。

程序13-1在合并的Account类中实现了存款以及用支票账户操作的属性。客户代码定义了Account对象，并且执行了相关的操作。这种客户代码是一个典型的前面向对象编程

（pre-object-oriented）标准，它反映了我们（经常是毫无理由）的信念，即人们常常可以正确地使用变量。

程序13-1 在相同的Account类中合并不同功能的例子

```
#include <iostream>
using namespace std;

class Account {

  double balance;                                  // for all kinds of accounts
  double rate;                                     // for savings account only
  double fee;                                      // for checking accounts only

public:

  Account(double initBalance = 0)                  // for checking accounts only
  { balance = initBalance; fee = 0.2; }            // use fee but not rate

  Account (double initBalance, double initRate)    // for savings
  { balance = initBalance;  rate = initRate; }     // no fee here

  double getBal()
  { return balance; }                              // common for both accounts

  void withdraw(double amount)                     // common for both accounts
  { if (balance > amount)
       balance -= amount; }
  void deposit(double amount)                      // common for both accounts
  { balance += amount; }

  void payInterest()                               // for savings accounts only
  { balance += balance * rate / 365 / 100; }
  void applyFee()
   { balance -= fee; }                             // for checking accounts only
  } ;

int main()
{
  Account a1(1000), a2(1000,6.0);                  // a1: checking, a2: savings
  cout << "Initial balances: " << a1.getBal()
       << " " << a2.getBal() << endl;
  a1.withdraw(100);  a2.deposit(100);              // no problem
  a2.payInterest();  a1.applyFee();                // no errors
  cout << "Ending balances: " << a1.getBal()
       << " " << a2.getBal() << endl;
  return 0;
  }
```

现在，我们不再盲目认为人不会出错。总会有人在某时某处输入某些字符。例如上述客户代码中的第5行可能被写成以下这种形式：

```
a1.payInterest();  a2.applyFee();  // miss takes a maid (joke)
```

我们当然无法避免所有的错误，（这就是为什么程序要测试的原因）但是应当尽量避免错误的产生，或者至少不需要计算实际的输出结果就知道程序有没有错误。因此，这个设计需

要改进。

注意，当一个账户创建时，客户代码对每个账户的特点都做了明确的注释。但是在程序设计中，只允许程序员用代码来表达他的想法而不是使用注释。为了把这个想法用程序实现，服务器类（本例中是Accout类）通过明确区分不同种类Accout对象的任务来满足客户代码的需要。

13.1.2　把保持程序完整性的任务推向服务器

为了避免客户代码不正确地使用服务器对象而带来的危害，我们可以给服务器类添加一个附加属性，比如通过一个标签域来描述一个实际的对象是属于哪一种Account类。这意味着我们把子类（subclass）加入到一个类中。

当一个对象被创建时，在对象的初始化过程中设定此标签域以指明该对象属于哪一个子类。当使用该对象（如进行payInterest()或applyFee()操作）时，检查此标签域来确保这样的操作对该种类的对象是合法的。

例如，当创建一个Account对象时，我们把标签域设为0来表示该对象用于支票账户；假如对象将用于存款账户，则把标签域设为1。这意味着构造函数应该知道正在创建的对象是哪个类型的Account对象。

在这个例子中，可以巧妙地利用这样一个事实：两个不同种类的账户的构造函数有不同数目的参数。但是，使用数字值设置标签域不符合软件工程的思想方法。因为只有代码的设计者知道0意味着支票账户，1意味着存款账户，其他人对此会感到迷惑不解。设计者怎样才能把他的设计思想（具体而言，在本例中即用哪一个标签取值来代表支票账户和存款账户的设计思想）传递给程序的维护人员呢？这也是为什么在C++中要使用枚举类型的原因，我们可以为Account类使用一个局部的枚举类型域Kind。因为Kind类不会被Account类以外的类使用，所以我们要把枚举类型Kind嵌入到Account类中。这个名字不会影响到全局名字空间，也不会妨碍项目中其他程序员在别的地方使用该名称。

```
class Account {
  enum Kind { CHECKING, SAVINGS } ;                    // constants for account kind
  double balance;
  double rate, fee;
  Kind tag;                                            // tag field for object kind
public:
  Account(double initBalance = 0)                      // checking account
  { balance = initBalance;  fee = 0.2;
    tag = CHECKING; }
  Account (double initBalance, double initRate)        // savings account
  { balance = initBalance;  rate = initRate;
    tag = SAVINGS;  }
  . . .  } ;                                           // the rest of Account class
```

假如运气不好，客户代码也要使用Kind类型，并且客户代码将明确地指明正在创建的是哪种账户。这意味着构造函数代码中要包含一个标识账户类型的参数。

为增加此例子的难度而使其更合乎实际，我们假定对所有的存款账户都用相同的利率，因此没有必要在客户代码中作特别声明。于是，Account类仅仅需要一个构造函数；同时，客户代码必须指明账户对象的种类；另外，Kind应声明为全局类型（这将使全局名字空间混乱，进而加大编程小组成员之间协作的工作量）。这个新的Account类如下所示：

```
enum Kind { CHECKING, SAVINGS } ;                    // constants for account kind

class Account {
  double balance;
  double rate, fee;
  Kind tag;                                          // tag field for object kind
public:
  Account(double initBalance, Kind kind)             // one constructor only
  { balance = initBalance;  tag = kind;              // set the tag field
    if (tag == CHECKING)
       fee = 0.2;                                    // it is checking account
    else if (tag == SAVINGS)
       rate = 6.0; }                                 // it is savings account
  . . . } ;                                          // the rest of Account class
```

注意在上述代码中，对于存款账户对象的利率和支票账户对象的支票兑换费，我们坚持没有使用同一个内存位置。如果一个应用要在内存中处理大量的Account对象，而且内存十分珍贵，就可以考虑使用同一个内存。否则这样做只会增加代码之间的相互依赖关系。因此，还是要尽量避免以不同方式使用同一内存的情况。

现在，客户代码使用枚举数据类型明确地指出了正在构造的对象属于哪种Account对象。这样一来，注释就显得多余了。它们仅仅重复了代码设计者已经在代码中表示出来的意思，代码已经将设计人员的意图传达给维护人员了。

```
Account a1(1000,CHECKING);        // a1 is checking account
Account a2(1000,SAVINGS);         // a2 is savings account
```

在上面的例子中，即使在客户代码需要使用这个类型的值时，枚举类型Kind也不会破坏名字空间，方法之一是在Account类中将其再次定义为局部类型。

```
class Account {
  double balance;
  double rate, fee;
  Kind tag;                                          // tag field for object kind
public:
  enum Kind { CHECKING, SAVINGS };                   // constants for account kind
  Account(double initBalance, Kind kind)             // one constructor only
  { balance = initBalance;  tag = kind;              // set the tag field
              if (tag == CHECKING)
                 fee = 0.2;                          // it is checking account
              else if (tag == SAVINGS)
                 rate = 6.0; }                       // it is savings account
          . . .  } ;                                 // the rest of Account class
```

当使用枚举类型的字面值作为构造函数的参数时，客户代码就必须使用类的作用域运算符。

```
Account a1(1000,Account::Kind::CHECKING);     // a1 is checking account
Account a2(1000,Account::Kind::SAVINGS);      // a2 is savings account
```

为了实现这种设计，Kind类型不能在Account类的私有部分定义，这一点与前一个具有两个构造函数的Account类是不一样的。要定义为Public，因为客户代码要访问该枚举类型Kind的取值（CHECKING、SAVINGS）。请注意在Account类里面使用该枚举变量（对于数据成员tag）时，并不一定要紧跟在该变量的定义之后。虽然C++编译程序是一个一遍扫描

的编译程序，但在类定义中却会进行两遍扫描。

然而，这只是在比较新的编译程序里面才能通过，一些老一点的编译程序可能会因为定义tag而没有定义Kind类型而中止编译，对这些编译程序而言，Kind类型的定义应该在tag域定义之前。为了在客户代码中能识别这个定义，Kind类型必须放在类定义的public部分。为了协调各方面的矛盾，可以在类定义中定义另外的public和private部分。

```
class Account {
  double balance;
  double rate, fee;
public:
  enum Kind { CHECKING, SAVINGS };                    // constants for account kind
private:
  Kind tag;                                           // tag field for object kind
public:
  Account(double initBalance, Kind kind)              // one constructor only
  { balance = initBalance;  tag = kind;               // set the tag field
    if (tag == CHECKING)
       fee = 0.2;                                     // it is checking account
    else if (tag == SAVINGS)
       rate = 6.0; }                                  // it is savings account
  . . .  } ;                                          // the rest of Account class
```

在构造函数中合理地初始化对象的tag域后，Account类的设计者能够保护客户代码免受不一致的影响。为了确保客户代码对一个存款账户对象调用withdraw()之后，不会错误地改变变量fee的值，服务器类（即Account类）必须检查存款对象的特征，并且只对支票账户收取处理费。

```
void withdraw(double amount)          // common for both accounts
{ if (balance > amount)
     { balance -= amount;
       if (tag == CHECKING)           // for checking accounts only
          balance -= fee; } }
```

正如我们所看到的那样，函数applyFee()的功能在成员函数withdraw()中也都具有了，这样编写客户代码的程序员就没必要一定要记住调用的是哪种类型的对象。希望大家在此能总结一下信息隐藏的概念和将任务推给服务器完成的程序设计思想。

同样，payInterest()方法检查消息发送的对象是否是一个存款账户。如果是，则支付当天的利息。如果该账户是一个支票账户，就会打印出一条运行时错误信息，通知测试人员，客户代码程序员错误地调用了错误对象的函数，因此取消该操作。

请注意这里使用的术语。是Account类的设计者代表客户代码工作。在前面向对象编程时期，客户代码必须保证自身的程序完整性（或确保没有错误）。在面向对象编程时期，我们把实现程序完整性的任务从客户代码推到服务器类。这是一个非常普遍的设计方法，值得我们善加使用。

程序13-2显示了Account类的实现，它应用了这个技巧保证客户代码行为的有效性。注意，Kind的类型是在Account类的外面定义的。图13-1显示了程序运行后的输出。

程序13-2　客户代码正确的运行时测试的例子

```
#include <iostream>
```

```
using namespace std;

enum Kind { CHECKING, SAVINGS } ;          // constants for account kind

class Account {
  double balance;
  double rate, fee;
  Kind tag;                                // tag field for object kind
public:
  Account(double initBalance, Kind kind)
  { balance = initBalance;  tag = kind;          // set the tag field
    if (tag == CHECKING)
       fee = 0.2;                                // for checking account
    else if (tag == SAVINGS)
       rate = 6.0; }                             // for savings account

  double getBal()
  { return balance; }                            // common for both accounts

  void withdraw(double amount)                   // common for both accounts
  { if (balance > amount)
      { balance -= amount;
        if (tag == CHECKING)                     // for checking accounts only
           balance -= fee; } }

  void deposit(double amount)
  { balance += amount; }

  void payInterest()                             // for savings account only
  { if (tag == SAVINGS)
     balance += balance * rate / 365 / 100;
    else if (tag == CHECKING)
     cout << " Checking account: illegal operation\n";  }
  } ;

int main()
{
  Account a1(1000,CHECKING);                     // a1 is checking account
  Account a2(1000,SAVINGS);                      // a2 is savings account
  cout << " Initial balances: " << a1.getBal()
       << "   " << a2.getBal() << endl;
  a1.withdraw(100);  a2.deposit(100);            // no problem
  a1.payInterest();  a2.payInterest();           // is this any good?
  cout << " Ending balances: " << a1.getBal()
       << "   " << a2.getBal() << endl;
  return 0;
  }
```

```
Initial balances: 1000  1000
Checking account: illegal operation
Ending balances: 899.8  1100.18
```

图13-1 程序13-2的输出结果

因为现在Kind类型是全局的，客户代码可以在构造函数中仅使用CHECKING和SAVINGS

标识指定账户的类型。

```
Account a1(1000,CHECKING);              // a1 is checking account
Account a2(1000,SAVINGS);               // a2 is savings account
```

这当然比我们以前的做法简单，那时Kind的类型是局部类型（因此要使用Account::Kind::CHECKING 和Account::Kind::SAVINGS）。

这样做写起来简单一些。但是前一个版本可以让程序的维护人员更加清楚这些枚举变量属于Account类，而不属于其他服务器类。这个版本写起来简单，但是设计人员必须与其他可能使用这个名字的设计人员协调合作，以统一全局名Kind的使用。正如第1章所写的一样，如果简明的代码导致协调工作的增加和理解代码难度的增加，现代程序设计方法更倾向较为冗长的而不是简洁的代码。但这并不意味着我们总是要编写冗长的代码，而是应该在简明与增加协调工作减少代码的可读性之间做出权衡选择。

随着把不同子类型的数据和操作合并到一个类中，每个服务器对象的方法都加强了其自身的合法的操作。系统不会再崩溃，即使有错也可以分级退出（或者至少有合理的运行时错误信息），然而，服务器需要额外附加的代码进行类型分析，根据每个给定对象的标签值，每个方法加强了独立于其他方法的合法操作。对一个含有多种对象类型的大型系统而言，就会有大量的依赖对象类型的方法，这使服务器代码变得笨拙。

Account类知道太多关于处理不同子类型对象（支票账户和存款账户）的与类无关的事件，这样会使设计和维护的信息量十分庞大。假如还需要添加这个对象的另一个种类（子类型），就必须扩展已有类的每一个方法。当这些代码的无关部分都受到影响时，就必须反反复复地做大量的测试。

这个处理方法的主要问题是客户代码的错误仍然是运行时错误，而不是编译时的错误。因此要耗费人力去分析这些错误信息的来源，去观察客户代码发生的改变。一个好的服务器类应该设计成：能够在不正确地使用了不同类型的对象时返回语法错误，而不是运行时错误。

13.1.3 为每种服务器对象建立单独的类

解决这个问题的一个比较好的方案是设计一组独立的类，让每一个类只完成某一特定的工作，而不包含所有对象子类的全部属性。在上述的例子中，意味着设计两个类：CheckingAccount类和SavingsAccount类。

我们重新设计了这些类。CheckingAccount类将只包含所有与支票账户相关的内容，而不必去考虑任何与存款账户相关的内容。

```
class CheckingAccount {
  double balance;
  double fee;                                   // no interest rate
public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }       // a checking account
  double getBal()
  { return balance; }                           // common for both accounts
  void withdraw(double amount)
  { if (balance > amount)
      balance = balance - amount - fee; }       // unconditional fee
  void deposit(double amount)
  { balance += amount; }
```

```
} ;
```

与此类似，SavingsAccount类将只包含所有存款账户所要涉及的功能，而不关心任何与支票账户相关的功能。

```
class SavingsAccount {
  double balance;
  double rate;                                    // no checking fee
public:
  SavingsAccount(double initBalance)
  { balance = initBalance;  rate = 6.0; }         // a savings account
  double getBal()
  { return balance; }                             // common for both accounts
  void withdraw(double amount)
  { if (balance > amount)                         // same interface, different code
      balance -= amount; }
  void deposit(double amount)                     // common for both accounts
  { balance += amount; }
  void payInterest()                              // for savings account only
  { balance += balance * rate / 365 / 100; }
} ;
```

程序13-3列出了实现这个方法的程序源代码。注意在代码中不再有枚举类型变量Kind，不再需要它，也不用再考虑它是全局参数还是局部参数。即使每种类型的账户都使用相同数目的参数来初始化，客户代码依然不需要枚举变量来指明正在创建的是什么类型的账户。为什么呢？因为客户代码明确地定义了账户对象a1、a2是分别属于CheckingAccount类还是属于SavingsAccount类。因此，每个对象定义都调用了合适的构造函数。程序的输出显示在图13-2中。

程序13-3 为对象的不同子类型设计各自的类的例子

```
#include <iostream>
using namespace std;

class CheckingAccount {
  double balance;
  double fee;                                     // no interest rate

public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }          // a checking account

  double getBal()
  { return balance; }                             // common for both accounts

  void withdraw(double amount)
  { if (balance > amount)
      balance = balance - amount - fee; }         // unconditional fee

  void deposit(double amount)
  { balance += amount; }
} ;

class SavingsAccount {
  double balance;
  double rate;                                    // no checking fee
```

```
public:
  SavingsAccount(double initBalance)
  { balance = initBalance;  rate = 6.0; }        // a savings account

  double getBal()
  { return balance; }                             // common for both accounts

  void withdraw(double amount)                    // common for both accounts
  { if (balance > amount)
      balance -= amount; }

  void deposit(double amount)
  { balance += amount; }

  void payInterest()                             // for savings account only
  { balance += balance * rate / 365 / 100; }
} ;

int main()
{
  CheckingAccount a1(1000);                      // a1: checking
  SavingsAccount a2(1000);                       // a2: savings
  cout << " Initial balances: " << a1.getBal()
       << "  " << a2.getBal() << endl;
  a1.withdraw(100);  a2.deposit(100);            // no problem
  //a1.payInterest();                            // this is a syntax error now!!
  a2.payInterest();                              // this is ok
  cout << " Ending balances: " << a1.getBal()
       << "  " << a2.getBal() << endl;
  return 0;
  }
```

```
Initial balances: 1000  1000
Ending balances: 899.8  1100.18
```

图13-2　程序13-3的输出结果

这种设计漂亮地解决了错误的客户使用所产生的问题。它产生的是编译错误，而不是运行错误。

```
a1.payInterest();          // syntax error: method not found
```

使用这种设计仅有的问题是它不能很好地把设计人员的意图传达给维护人员。在这里，我们看到两个类有许多共同点：balance数据成员、withdraw()和deposit()操作，以及对数据的访问，但是设计本身却没有指明这两个类有共同的东西。这些类的设计者知道它们有这些共同的特征，但是这些信息并没有传达给程序的维护人员。

当然，这些类有相似的名字，但是在一个大程序里只有相似的名字却是不够的。在程序13-3中，两个类被放在同一页（并且在同一个源文件）中，但在实际工作中它们可能是分开的，它们的相似性也可能会被维护人员忽视。当这些类中的一个被修改时，并不能保证另一个类也被修改。当这组不同类型的对象的数目增长时，这些类的共同特征并不能标记出来。通常，源代码中不一定能够表达出程序设计者的思想。

13.1.4 使用C++的继承去链接相关类

继承是这个问题的另一个解决方案。我们可以创建一个类，它包含了对所有子类型共有的特点的通用命名。以面向对象分析和设计的术语来说，即这个类泛化了这些子类所具有的状态和行为。每个特殊类都把自身特有的特征加到泛化的类中。

例如，一个账户的概念相对于存款账户和支票账户这两个特殊的概念而言，是一个概括性的概念。Account类并没有把存款账户和支票账户的全部特征合并在一起，而是仅包含了这两个不同类型账户（CheckingAccount和SavingsAccount）所公共的特征，这些公共的特征包括数据成员balance、方法getBal()、withdraw()和deposit()。

```
class Account {                          // base class: common features
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
  double getBal()
  { return balance; }                    // common for all accounts
  void withdraw(double amount)           // common for all accounts
  { if (balance > amount)
  balance -= amount; }
  void deposit(double amount)
  { balance += amount; }
} ;
```

上述代码中的类与前面所看到的C++类相比，仅有的区别是用关键字protected代替了原有的关键字private。关键字protected的作用与关键字private相似，也是用于阻止从类的外部访问类的成员（元素）。但两者最重要的不同点在于：关键字protected允许该类的继承类访问。

在C++的术语里，我们把这个概括了其他类的特征并且合成它们共同特征的类叫做基类（base class），它作为进一步继承的基类。向基类中指定的共同特征添加了新特征的特殊类称为派生类（derived class）。派生是C++中对继承的术语，Java就使用术语扩展（extension）而不是“派生”。

基类的其他较流行的术语是超类（superclass）和父类（parent class）。与此对应，派生类就叫做子类（subclass）和孩子类（child class）。在基类作为一个数据类型的上下文中，把派生数据类型叫做子类型（subtype）更为合适。

派生类对一个更一般化的基类中的一些特征进行增加或替换。派生类中增加的数据和方法就反映了类之间的特化关系。

例如，CheckingAccount类和SavingsAccount类可以设计成泛化类Account的两个独立的特化类，Account类实现了它们的共同特征。它们增加了与兑换费和支付利率相关的功能，这是泛化类Account所不具有的。

派生类SavingsAccount把数据成员rate和成员函数payInsterest()增加到基类Account中。它使用基类的数据成员balance和成员函数getBal()、withdraw()及deposit()，没有替换基类的任何特征。下面的代码显示了定义一个派生类时所必需的工作。我们不用在派生类中重复从基类继承而来的特征，在派生类中仅仅需要描述的是派生类增加到基类中的特征，或者是派生类替换的基类某些特征（将在下一节讨论继承的语法）。

```
class SavingsAccount : public Account {          // derived class
  double rate;
public:
  SavingsAccount(double initBalance)
  { balance = initBalance;  rate = 6.0; }        // savings account
  void payInterest()                             // not for checking
  { balance += balance * rate / 365 / 100; } } ;
```

派生类CheckingAccount增加了数据成员fee到Account类中，它使用基类的数据成员balance和成员函数getBal()、withdraw()及deposit()，但用它自己的withdraw()函数替代了基类的成员函数withdraw()。这个函数用来计算兑换费（而不像基类中的成员函数withdraw()那样）。

```
class CheckingAccount : public Account {         // derived class
  double fee;
public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }         // checking account
  void withdraw(double amount)
  { if (balance > amount)
      balance = balance - amount - fee; }        // not for savings
} ;
```

这样，继承成为程序开发阶段代码重用和设计重用的工具。这时不需要在每个特殊类中都重复Account类的共同特征，这些特征只要在基类中定义一次即可。它使设计更加紧凑（没必要再重复共同特征），而且加快了程序员的开发进度。

在这些例子中，账户、雇员以及库存信息等只是代表着一个抽象的概念，而并非那些需要在应用程序中建模的真实世界中的物体。毕竟，笼统的账户、雇员、以及库存信息这些本身都是不存在的，真正存在的只有支票账户、存款账户、螺母、螺丝钉以及正式员工和临时工等。

然而，在真实世界的实体之间却通常存在着“自然”的超类/子类关系，它们可以在代表这些实体的类之间的关系中体现出来。例如，每一辆小汽车是一种交通工具，而每一辆微型小客车又是一辆小汽车。这种关系能利用继承来表达。

继承可以是直接或者间接的。交通工具是一个小汽车的直接超类或者说是直接基类，小汽车是微型小汽车的一个直接超类或者说是直接基类。汽车是微型小汽车的间接超类或者说是间接基类。

如果一个类（如，小汽车）是另一个类（如，交通工具）的一个派生类，但同时又是第三个类（如，微型小汽车）的一个基类，这种情况也是允许的。

继承也能用于更进一步的程序开发。当需要执行更多特殊的操作时，一个派生类可以定义为仅仅提供这些新操作的类；而其他的操作依然像以前一样由基类来提供。

和类之间的一些任务分配一样，在软件开发中继承可以作为一个劳动分配的工具。一个巨大的类只能由一个开发者来开发，而基类和派生类可以由几个不同的程序员来共同开发，或者由一个程序员在不同的时间来开发。

除了作为提高抽象度的好方法之外，在类中使用共同特征还可以减少代码的编写量，提高劳动分工的模块化。但我们不能保证在实际编程中使用继承后总能减少代码的编写量。如果基类比较小，只有几个子类型，这样即使使用继承，源代码也不会减少很多。假如基类比

较大，且有大量的不同子类型，而每个子类型仅仅需要增加很少的功能，那么的确可以减少源代码量，因为没有在每一个类中都重复基类的代码。

在为派生类CheckingAccount类和SavingsAccount类编写代码时，冻结基类Account的代码，这是一个项目管理的强有力的范例。如果Account类在今后有所改变，那么这种改变也会自动地反映到所有派生类中（这可能有好有坏，但这是另外的问题）。

继承的另一常用情况是在运行时绑定方法，运行时绑定的其他术语是动态绑定和虚函数的多态性。很多人认为面向对象编程就是关于继承和多态性，但并不是这样的。

多态性是面向对象编程的一个特殊情况，在这种情况下程序要处理一组相关的对象，但是对不同类的对象执行相似而不完全相同的操作。这些不同类型的对象十分相似，因而能从一个共同的基类派生而来（如椭圆、长方形、三角形等都可以从图形类中派生而出）。而且那些操作如此相似以至于可以在每个类中取相同的名字（如draw()）。

多态性允许我们处理一连串的对象，它向每个对象都传送相同的消息，而不管这个对象属于哪个特殊的类。每种情况下实际调用的函数依赖于每个对象所属的基类，尽管看起来该函数调用形式上是调用了基类函数（虚函数）。听起来有点糊涂，是吗？不要紧。继续看下去就会成为多态性方面的专家。

13.2 C++继承的语法

使用C++继承的核心是派生类名字后面紧跟的冒号。它表示此处说明的是其基类的名字，以及继承模式的描述符——public、private还是protected。

程序13-4与程序13-3一样，但却是使用继承来实现的。这里的客户代码是程序13-3中代码的扩展，因此，这个程序的输出（见图13-3）也是程序13-3输出的扩展。

程序13-4 Account类的继承层次的例子

```
#include <iostream>
using namespace std;

class Account {                                    // base class of hierarchy
protected:
  double balance;

public:
  Account(double initBalance = 0)
  { balance = initBalance; }

  double getBal()
  { return balance; }                              // common for both accounts

  void withdraw(double amount)                     // common for both accounts
  { if (balance > amount)
  balance -= amount; }

  void deposit(double amount)
  { balance += amount; }
} ;

class CheckingAccount : public Account {           // first derived class
  double fee;
```

```
public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }          // for checking account

  void withdraw(double amount)
  { if (balance > amount)
      balance = balance - amount - fee; }         // unconditional fee
} ;

class SavingsAccount : public Account {           // second derived class
  double rate;

public:
  SavingsAccount(double initBalance)
  { balance = initBalance;  rate = 6.0; }         // savings account

  void payInterest()                              // not for checking
  { balance += balance * rate / 365 / 100; }
} ;

int main()
{
Account a(1000);                                  // base class object
CheckingAccount a1(1000);                         // derived class object
SavingsAccount a2(1000);                          // derived class object
a1.withdraw(100);                                 // derived class method
a2.deposit(100);                                  // base class method
a1.deposit(200);                                  // base class method
a2.withdraw(200);                                 // base class method
a2.payInterest();                                 // derived class method
a.deposit(300);                                   // base class method
a.withdraw(100);                                  // base class method
//a.payInterest();                                // syntax error
//a1.payInterest();                               // syntax error
cout << " Ending balances\n   account object:          "
     << a.getBal()<<endl;
cout << "   checking account object: " << a1.getBal() << endl;
cout << "   savings account object:  " << a2.getBal() << endl;
return 0;
}
```

```
Ending balances
  account object:          1200
  checking account object: 1099.8
  savings account object:  900.148
```

图13-3　程序13-4的输出结果

13.2.1　基类的不同派生模式

用来标识继承模式的关键字与标识类成员访问权限的三个关键字是完全一样的：public、protected和private。正是使用这些关键字（连同前面的冒号）来指明类之间的继承关系。

既然这些关键字是相同的，许多C++程序员认为每个关键字的意义都与对类元素访问的控制意义相同。例如，在下面这小块程序中，关键字public就用了两次。

```
class CheckingAccount : public Account {  // Account is the base
  double fee;                             // data member added in derived class
public:                                   // start of public segment of the class
   . . . } ;                              // the rest of derived class CheckingAccount
```

千万不要以为这两种情况下关键字public的意义是一回事，它们完全不同。其仅有的共同点是关键字public和冒号，除此之外，没有其他的关系。在访问权限的public关键字中，冒号在关键字的右边，它意味着跟在它后面的类成员可以从程序的任何地方访问。在继承模式的关键字中，冒号在关键字的左边，这意味着对继承类的类成员的访问权限与在基类中是一致的，如基类中私有的部分在派生类中依然私有等。

的确，我们使用了同一个关键字public来设定对类的数据成员的访问权限和派生类的访问模式。但相同的只是关键字，其他的并不相同。

在继承模式中使用冒号和关键字，在语法上就将基类和派生类连在了一起。无论源代码中类的定义放在哪里，检查派生类定义的维护人员都可以有一个无二义的可视化线索，这个线索确定了两件事情：

1) 存在另一个类作为这个类的基类。

2) 基类的名字。

如果用一个统一建模语言（UML）图形来表示，则两个类之间的关系可以表示为类图标之间的连接线，线的一端有指向基类的空心三角形。假如基类有不止一个派生类，那么每个派生类都可以用一个空心三角形连到基类，也可以通过一个共同的空心三角形连到基类。图13-4显示了用两种不同的方式描述Account类和它的两个派生类之间的关系。

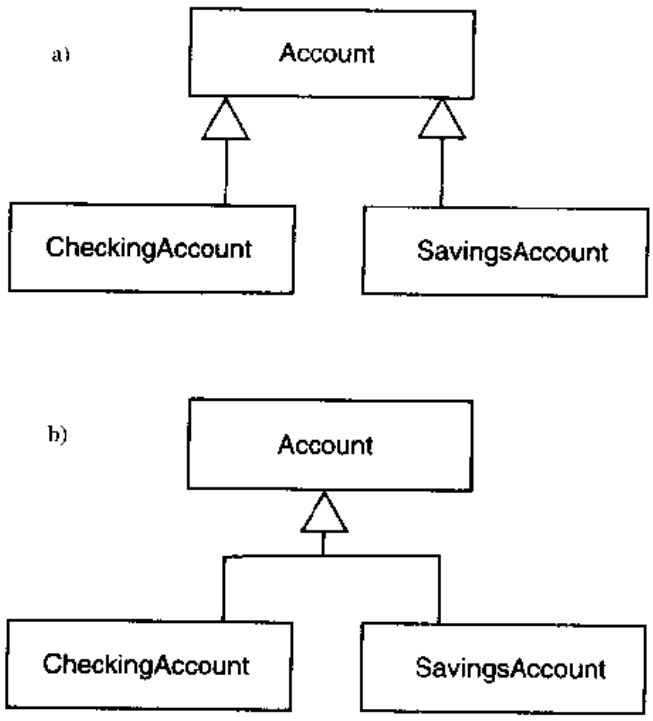


图13-4 在Account类层次中类的关系

这是一个利用继承去组织相关应用概念的例子。一个支票账户“是一种”账户：即每个支票账户是一个账户，但并非每个账户都是支票账户。这只是关于继承关系的一般情况。再如，一辆小汽车“是一种”交通工具，每辆小汽车都是一种交通工具，但并非每种交通工具都是一辆小汽车。同样，一个矩形“是一种”多边形，即每个矩形都是多边形，但不是每个多边形都是矩形。

正是存在“是一个”关系，才在概念上将类连接起来，并且适合于使用继承来表达。它不同于用“有”关系连接对象的聚集。例如，一个矩形“有”一些顶点，一个历史记录对象“有”标本对象。但如果说一个历史记录对象“是一个”标本对象就是不正确的，因为这两种对象有完全不同的数据成员和完全不同的行为。在继承的情况下，两个类的数据和行为也不相同，但它们有一个在基类中定义的共同子集。Account类有一个数据成员balance和一个方法deposit()。根据继承的优点，即使这些元素没有在CheckingAccount类定义时列出来，CheckingAccount类也仍有一个数据成员balance和一个deposit()方法。

这是继承的要点：继承是一种类之间的关系。Account类定义了数据成员balance，因而，CheckingAccount类就不需要再如此定义。既然CheckingAccount类是从类Account继承而来的，那么在内存中CheckingAccount对象也就有一个与Account对象相似的数据成员balance。一个CheckingAccount对象也是一个Account对象，它具有Account对象的所有属性甚至更多：有一些是CheckingAccount类定义的属性。

因此，继承并不能节约存储空间。每个CheckingAccount对象都有Account的全部数据。当类变得太大时，继承有助于创建比较小的类，同时指明存在于这些小的类之间的逻辑连接。例如，程序13-4就显示了CheckingAccount类和SavingsAccount类的关系——它们都从Account类继承而来，但程序13-3却不能指明这种逻辑连接关系，它的类定义在程序源代码中是放在一起的，但它们却并没有强调共同的数据和成员函数，必须由我们自己推理出来。

每个C++类在派生时都可以作为基类。继承的层次是可以传递的，例如，TradingAccount类可以定义为CheckingAccount类的派生类。一个TradingAccount对象就可以拥有CheckingAccount对象的所有功能，而既然一个CheckingAccount对象又具有Account对象的所有功能，那么一个TradingAccount对象就具有了一个Account对象的所有功能。

从这个观点来看，通常用超类和子类这两个术语来代表继承关系中的基类和派生类，并不十分准确。它们表明从某种角度来说，基类或者说超类要比派生类或子类更高级一些，但事实上却不是如此。

处于派生类结构层次底部的派生类并没有失去基类的功能，如CheckingAccount对象能比Account对象做更多的事情，而TradingAccount对象不但能完成CheckingAccount对象能完成的每一个功能，甚至还可以完成更多的功能。只是对成员关系的限制朝着层次结构的底部方向而增加。一个CheckingAccount是一个受限的Account类，即CheckingAccount对象肯定少于Account对象，因为每一个CheckingAccount对象也属于Account对象。类似地，TradingAccount对象也少于CheckingAccount对象，也是因为每一个TradingAccount对象都属于CheckingAccount对象。

顺着层次结构往底层看，我们会看到每个子类的对象实例越来越少，但这些子类对象的

可用功能却越来越多。从数学的观点来看，一个集合中的实例数量可能是重要的。而从编程的观点来看，一个对象所能提供的服务的数量才是重要的。超类比子类提供的服务更少。这也是我们不喜欢使用超类、子类这些术语的原因，因此用基类、派生类会更好。

继承强化了模块化和代码重用。一组设计良好的通用类可以组织成一个库。库中类的接口可以发布出来，但是其实现是封装的。通过创建新的派生类可以定制库中的类。这些类把一些新的数据成员和新的成员函数加入到基本类库之中。这是创建图形用户界面的通用技术。应用程序中的类就是从一些类库中的类（窗口、对话框、命令按纽）继承而来。编写应用程序的程序员可以使用类库中已有的功能，只用加入一些特定的功能——例如说明这个特定的窗口、对话框或者命令按纽看起来是什么样子的，在应用中有什么行为等。

在客户定制的过程中没必要改变类库中的基类。因此，基类不用编辑修改也不用重新编译。

程序13-4说明了每个派生类必须明确地指出它的基类；它可以增加一些数据成员或成员函数，这是可选的。

```
class SavingsAccount : public Account {     // syntax of derivation
  double rate;                              // additional feature
public:
 . . . } ;                                  // the rest of SavingsAccount
```

然而，客户代码没有必要了解派生的有关情况。如果客户代码在独立的文件中实现，那么这些文件只需要了解派生类而不是基类。在定义和实现派生类的文件中才需要知道基类。这进一步证明继承不是更好地为客户代码服务的机制，而是用来设计服务器类（在本例中是SavingsAccount和CheckingAccount类）的机制。不管这些类是怎样设计的——用继承还是从头开始设计，对客户代码来说都没什么不同。

13.2.2 定义和使用基类对象和派生类对象

当客户代码需要一个对象时，它可以定义并使用基类对象和派生类对象，如果客户代码在单独的文件中，那么还应该包括每个类（在程序13-4中，就是基类Account和派生类SavingsAccount和CheckingAccount）的头文件。

应该调用哪个方法去响应一个消息呢？这个方法应该根据目标对象声明的类型来定义。编译程序找到目标对象的定义，并搜索目标对象所属类的定义。程序13-4显示了在客户代码中的典型情况，我们必须能区分这些情况。

```
Account a(1000);                            // base class object
CheckingAccount a1(1000);                   // derived class object
SavingsAccount a2(1000);                    // derived class object
a1.withdraw(100);                           // derived class method
a2.deposit(100);                            // base class method
a1.deposit(200);                            // base class method
a2.withdraw(200);                           // base class method
a2.payInterest();                           // derived class method
a.deposit(300);                             // base class method
a.withdraw(100);                            // base class method
//a.payInterest();                          // syntax error
//a1.payInterest();                         // syntax error
cout << " Ending balances\n   account object:
```

```
        << a.getBal() << endl;
cout << "    checking account object: " << a1.getBal() << endl;
cout << "    savings account object:  " << a2.getBal() << endl;
```

假如目标对象属于基类，编译程序就对属于基类的成员函数发出一个函数调用。

```
a.deposit(300);                    // base class method
```

即使该方法也在派生类中定义了，而且其操作与派生类对象的操作不同，这个规则也是有效的。例如，withdraw()方法就定义为不同于派生类CheckingAccount的方法。当消息的目标是一个基类对象时，调用的仍然是基类的withdraw()方法。

```
a.withdraw(100);                   // base class method
```

通常，在客户代码中，基类对象的表现行为就如同派生类并不存在一样。基类对象不能响应在派生类中定义的消息，除非这些消息是从基类中继承来的功能。例如，试图要求一个Account对象去完成指定给派生类SavingsAccount对象的工作时，编译程序将会拒绝。

```
a.payInterest();                   // syntax error
```

即使通过继承关系，Account和SavingsAccount类是彼此相关的，但这仍不足以让Account对象响应派生类的消息。payInterest()方法不是在Account类中定义的，所以这个函数调用将会产生语法错误。

然而，当消息的目标是一个派生类对象时情况有些不同。我们应该能区别以下三种情况：

1) 该方法是从基类继承而来并且在派生类没有重新定义。

2) 在基类中并没有该方法，而只是派生类扩充定义的。

3) 该方法属于基类，但派生类又对它进行重新定义。

当客户代码调用一个继承而来的方法时，编译程序会有一个问题。与处理其他消息相似，编译程序找出消息的目标对象的类型（这是一条发送给派生类对象的消息），并且在派生类的规格说明中搜索指定的成员函数名。

```
a1.deposit(200);                   // base class method
```

显然，成员函数并不存在，因为这个继承来的方法（本例中，即deposit()）只在基类中有描述，而没有在派生类中描述。语法上，在派生类中再次对继承而来的方法进行描述也是可以接受的，但它将是一个重新定义了的方法，而不是原来那个继承而来的方法。

当方法在目标对象所属的类中找不到时，编译程序会警告客户端代码程序员调用的方法不存在。在这样做之前，编译程序检查在类的定义中类名后是否有一个冒号。如果有，编译程序将得出这是一个派生类的正确结论，然后找到基类名，进而搜索到基类的定义。如果能找到所要的方法当然很好，但如果找不到该方法，编译程序将检查这个类是否也有一个基类，并且重复这个过程，直到满足以下两个条件中的一个为止：在继承链中找到一个没有基类的类，或者在类的定义中找到这个成员函数。如果是后一种情况，编译程序将检查调用函数的参数个数及相应类型是否与函数定义一致，再为这个函数调用产生目标代码。

注意，继承的使用打破了面向对象编程的第一个原则：在类的作用域范围内把数据和操作绑定到类定义中。当然，继承的使用作为一个编程技巧，对面向对象程序设计而言是非常重要的。因此，程序员在实际编程中不要受一些抽象原则的局限。C++在概念和技术两个方面进行了调整，以便在程序员的需要和原则之间进行调和。

在概念层面上，C++声明一个派生类对象是基类的一个对象（加上更多其他特点），因此，派生类有在基类中定义的所有数据和方法。在技术层面上，C++使派生类的作用域范围嵌套在基类的作用域范围内。根据我们已知的作用域规则（文件、函数、块及类作用域），派生类可以访问基类的方法

这的确令人混淆，但我们不用担心概念和技术问题。只需要记住，当编译程序无法在派生类中找到指定的方法时，它会在基类中查找指定的方法。在这一章的后面部分，我们将用一个单独的小节去讨论继承中的作用域规则和名字解析规则。

在第二种情况下，方法的目标是一个派生类的对象，但是该方法在基类中不存在而在派生类中存在，这种情况比较简单。我们可以运用标准的规则去解释一个函数调用。编译程序在派生类的定义中找到这个方法，然后停下来。假如参数与函数标识不匹配，这就是一个语法错误。如果参数匹配，将产生合适的函数调用。

```
a2.payInterest();                    // derived class method
```

类似的规则也可以应用于第三种情况中被派生类重新定义的方法。编译程序忽略继承关系的存在。就像我们在前面看到的那样，当一条消息的目标是一个基类对象时，编译程序将在基类中搜索合适的方法而忽略派生类。当消息目标是一个派生类的对象时，编译程序搜索指定的派生类，直到发现该方法才停止。显然，肯定可以找到该方法，因为它在派生类中重新定义了。

```
a1.withdraw(100);                    // derived class method
```

假如实际的参数个数以及相应类型与该方法的声明相匹配，编译程序将产生合适的函数调用。假如不匹配，将是一个语法错误。编译程序不会再到基类中去搜索一个可能更匹配的方法，就像我们马上要看到的那样，这可能是一个麻烦的根源。

13.3 对基类和派生类服务的访问

通常，一个派生类也“是一个”基类，派生类的每个对象都有基类的所有数据成员和成员函数，此外还加上派生类自己增加或重新定义的数据和函数。

从某种意义上看，派生类是基类的一个客户，就像任何C++代码都可以作为它的服务器类的客户一样。客户代码使用服务器类的服务：数据成员和成员函数。服务器类并不了解它的客户类，它甚至不知道客户类的名字。这很正常，因为服务器类或函数可能来自于一个库，它们也许在客户代码生成之前的若干年就已经编写了。客户类必须知道它的服务器类的名字和适合于客户类使用的公共服务的名字。

例如，程序13-4中客户代码显式地使用类名定义了一个Account对象，然后，客户代码就可以通过使用它们的名字访问Account的服务。

```
Account a(1000);                          // base class object
a.deposit(300);                           // base class method
cout << " Ending balances\n   account object:          "
     << a.getBal() << endl;
```

在这个例子中，Account类并不知道客户代码怎样使用它。就像我们说过的那样，相对于客户代码而言，Account类可能是在几个月（或几年）前由不同的程序员设计的。

类似地，派生类使用基类的服务（数据成员和成员函数），而且基类也不知道派生类，因

为在编程中服务器对象并不知道它的客户的标识。但派生类必须知道它的基类的名字，而且还要知道基类中适合派生类使用的非私有服务的名字。

例如，程序13-4中的派生类通过在冒号运算符之后指定基类的名字，建立了与基类Account的继承关系。

```
class SavingsAccount : public Account {    // syntax of derivation
  double rate;
public:
 . . . } ;                                 // the rest of SavingsAccount
```

复合（聚集）的客户-服务器关系与继承的派生类-基类关系是不同的。在复合关系中，客户代码必须实例化一个服务器对象去访问服务。在继承关系中，派生类并不一定要实例化一个独立的基类对象实例，在派生类定义中出现的基类名字就足够了。

在类复合（我们在前一章做了详细讨论）关系中，容器类并不为它的客户提供它的每个元素的服务；它提供的仅仅是在它自己的界面中明确列举出来的服务。例如，Point类，我们将它作为Rectangle类的一个元素，Point类有公共方法set()、get()和move()。

```
class Point {
  x, y;  // private coordinates
   public:
     Point (int a, int b)                  // general constructor
       { x = a;  y = b; }
     void set (int a, int b)               // modifier function
       { x = a;  y = b; }
     void move (int a, int b)              // modifier function
       { x += a;  y += b; }
     void get (int& a, int& b) const       // selector function
       { a = x;  b = y; } } ;
```

这并不意味着有Point类这一数据成员作为元素的Rectangle类能为它的客户提供同样的服务。下面是客户代码的一个例子。

```
Point p1(20,40), p2(70,90);    // top-left, bottom-right corners
Rectangle rec(p1,p2,4);        // composite object: client of Point
rec.set(30,40);                // this does not make sense
rec.move(10,20);               // this is ok: why the difference?
```

这里，方法set()和move()的不同之处在于Rectangle类不用再实现成员函数set()，但要按照Rectangle类上下文中的意义定义move()方法。

```
class Rectangle {
  Point pt1, pt2;                          // top-left, bottom-right corner points
  int thickness;                           // thickness of the rectangle border
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                 // move both points
  void setThickness(int width = 1);        // change thickness
  bool pointIn(const Point& pt) const;     // point in rectangle?
   . . . . } ;                             // the rest of class Rectangle
```

然而，一个派生类为它的客户提供的是它的基类的服务，类设计人员可以什么也不做就使其变为可能。例如，考虑程序13-4中的SavingsAccount类。

```
class SavingsAccount : public Account {            // another derived class
```

```
  double rate;                                          // added components
public:
  SavingsAccount(double initBalance)
  { balance = initBalance;  rate = 6.0; }               // for savings account
  void payInterest()                                    // for savings account
  { balance += balance * rate / 365 / 100; } } ;
```

根据该类的定义，这个类的客户代码可以定义SavingsAccount类的对象，并发送payInterest()消息给这些对象。但是，如果检查程序13-4中的客户代码，将可以发现SavingsAccount类的对象不只发送了这个消息。

```
SavingsAccount a2(1000);                    // derived class object
a2.deposit(100);                            // base class method
a2.withdraw(200);                           // base class method
a2.payInterest();                           // derived class method
cout << "    savings account object:  " << a2.getBal() <<endl;
```

客户代码使用的服务deposit()、withdraw()和getBal()，并没有在派生类SavingsAccount中列出来，它们仅仅只在基类Account中列出来，这在编译时不会有任何问题。编译程序顺着类定义中的继承链，很容易就可以找到在基类（或在基类的基类，或其他地方）中的这些成员函数。但客户端代码程序员要做什么呢？客户端代码程序员怎样知道这些服务对定义在客户代码中的对象是有效的呢？客户端代码程序员（以及维护人员）不得不去做编译程序所要做的事情：顺着类定义中的继承链去查找。

使用SavingsAccount类服务的客户端代码程序员（和维护人员）必须在Account类中查出对SavingsAccount类对象有效的功能。在程序13-4中，为了方便，我们把这些类的定义都放在一起，但对大系统和复杂的继承层次（一个派生类又作为另一个类的基类，并且另一个类又作为第三个类的基类等）而言，这是不可能的。对客户端代码程序员（和维护人员）而言，去查找派生类所提供的全部功能列表是一件繁琐的事情。因为仅仅有派生类的描述还不够，还要在其他地方查找。

这增加了设计的复杂性，错误很难被发现，甚至更难纠正。另外，继承的使用违背了面向对象编程的原则。继承对一个在继承层次体系里设计类的程序员而言是很方便的，是一个使程序设计重用和减少代码量的技巧。

对客户端代码设计者而言，两个独立的类（SavingsAccount类和CheckingAccount类）代表了非常好的工程化解决方案。它们把相关的数据和服务绑定在一起。当试图将一条消息送往一个错误的类时，编译程序会将它标识为错误。继承给这个解决方案添加了什么？两个类的共同数据成员和方法只需实现一次，而对基类的改变会自动传播到所有的派生类，这大大方便了服务器类的实现者。

继承使其他人理解服务器对象的功能时感到更加困难。一些C++类库提供的类包括大量的服务（超过100个），并且把这些服务分布到五层以上的继承中。为了弄清一个库中的窗口类能完成什么工作，我们必须学习所有的这些继承层次，而且当从类库的一个版本转换到另一个版本时，继承层次结构本身以及可用的服务都会发生改变，这就让工作更加困难。因此我们必须不停地学习以适应新的技术。C++编程永远不会是单调乏味的工作，特别是当我们肆意使用继承时。

与继承而来的特征不一样，重新定义的特征在派生类的服务列表中是直接可用的。我们

不需要在其他地方查找它们。它们通常都要完成与基类中所定义的服务一样的功能，但完成得更有效，或者在某种程度上使用了不同的数据或算法。

在程序13-4的继承例子中，派生类CheckingAccount重新定义了在基类Account已经定义的成员函数withdraw()。

这个重新定义的函数与基类的函数相比，完成了不同的工作，它使用了只在派生类中可用而不能在基类中使用的数据（数据成员fee）。通常是因为其他的派生类（本例中是SavingsAccount类）没有使用这个数据成员而发生这种情况。假如其他派生类也使用了这个数据（例如，在本例中，所有派生类都使用基类数据成员balance），那么数据成员应该放在基类中（如同程序13-4一样）。

在派生类重新定义的成员函数中增加数据是一个流行的设计技巧，它很通用但并不强制使用。

派生类对象可以看作是派生类的所有组成部分（public、private和protected成分）以及基类的所有组成部分（public、private和protected成分）的总和。分配给派生类对象的内存也是分配给基类组成部分和派生类组成部分的内存的总和。

例如，在一台机器里，一个Account对象的大小是8个字节，而每个SavingsAccount类对象和CheckingAccount类的对象都是16个字节。假如作为数据成员的数据类型需在内存中对齐，那么要给该对象增加一些内存空间才能保证这些对象的数据成员正确地对齐。

派生类对象的客户可以使用派生类对象来调用基类的公共服务，就如同这些服务属于派生类本身的公共部分。例如，一个CheckingAccount类的对象会响应消息deposit()和getBal()，好像它们是在CheckingAccount类中定义的一样，客户代码并不知道（也不应该知道）有什么分别。

基类的成员不能访问在派生类中增加或重新定义的特征，基类对象中没有在派生类里面描述的数据成员和成员函数。例如，Account类不能访问在派生类SavingsAccount中定义的私有数据成员rate和公共成员函数payInterest()。下面这行语句是没有意义的：

```
Account a(1000);  a.payInterest();              // syntax error
```

本书认为这些语法规则很直观，它们拓展了认为派生类对象就是一个基类对象加上一些其他东西而成的想法。

就基类对象而言，它们并不知道与另一个类有关的服务，即使这个类从基类派生而来，毕竟它是另一个类。基类的对象不能响应其类定义中不存在的消息。

类似地，使用Account类的友元函数或友元类时，也不允许这个函数或者类直接访问它的派生类SavingsAccount类和CheckingAccount类的非公共元素。

13.4　对派生类对象的基类成员的访问

现在问题变得不再直观而是更加复杂了。一个派生类的成员和友元可以访问该派生类的所有数据成员和成员函数，它们也可以接受一些访问基类的数据成员和成员函数的请求。但它们只能访问基类的公共和受保护成员，而不能访问基类的私有数据成员和成员函数。它们也不能访问从同一个基类派生出的其他派生类的成员。

看待这个规则的一个方法是认为基类有三种类型的客户（或者说三个访问区域）。在内核区的是类成员函数和友元类，它们拥有对数据成员和成员函数的最大访问权限。它们可以访

问基类的公共的、受保护和私有数据成员和成员函数。它们具有这样的访问权限，是因为作为成员或者友元被声明在类的括号中。在中间区域的是派生类的成员函数和它们的友元。它们可以访问公共的和受保护类成员但不能访问私有成员，它们具有这样的权限，是由于在类定义时把这个类（直接地或间接地）声明为基类。

在本书中将外围访问区域的内容称为客户代码。正如我们所知道的，客户代码只能访问基类的公共数据成员和成员函数。客户代码获得访问基类的服务权限是由于它使用了基类作为一条消息的目标。有三种方式可以让对象为客户代码所使用：可以通过对象定义创建，也可以在堆内存中动态创建，或者将该对象（或对象的引用，或对象的指针）作为一个函数的参数。图13-5显示了类和它的三个访问区域的关系。

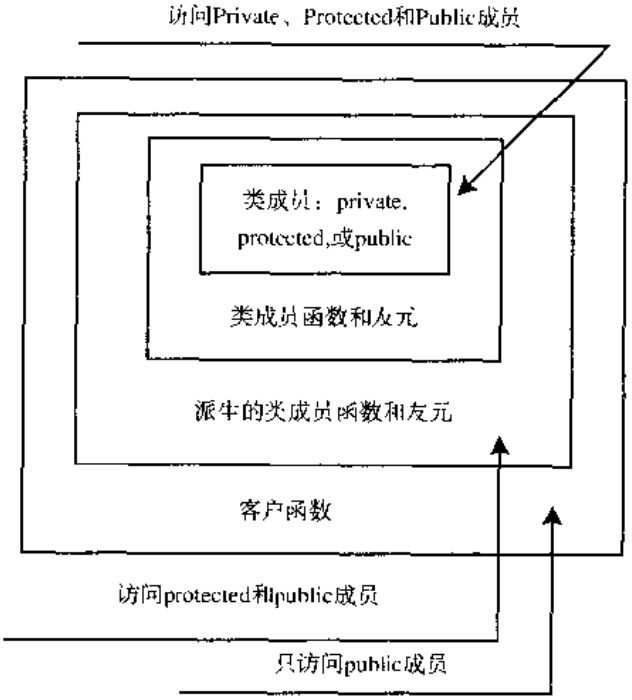


图13-5 一个类本身的成员和友元、派生类及客户代码可访问的区域

注意，只有在外围访问区域，客户代码才可以通过一个独立的服务器对象来访问数据成员和成员函数。在其他的两个区域，客户代码可以访问同一个对象的数据成员和成员函数，在内核区，此对象就是基类对象；在中间区域，此对象就是派生类对象。

根据派生模式可以改变中间区域的实现情况。基类成员能改变它们在派生类对象中的访问状态。基类中的公共成员可能会变成派生类中的受保护的甚至私有的类成员。在基类中的受保护成员可能变成派生类对象中的私有成员。

13.4.1 公共继承

每个基类都可以以私有、受保护或公共模式进行继承，这些模式定义了派生类中对基类元素的访问状态。用公共模式来继承，访问的状态仍与基类保持相同。在基类中的私有、受保护或公共元素在一个派生类对象中依然为私有、受保护或公共的。这是限制最少的情况——没有任何改变。

这样，派生类的方法可以访问一个派生类对象的基类的受保护和公共成员。图13-6显示了这种关系。图13-6中有一个派生类对象，它包括基类部分和派生类部分，每个部分都有公共的、受保护、私有成员。同时，也显示了将派生类对象作为其服务器类的客户代码。在图13-6中可以看到，客户代码可以访问基类的公共服务（数据成员和成员函数）和派生类的公共服务。对客户代码而言，派生类对象包含了在基类和派生类中定义为公共的功能的总和。

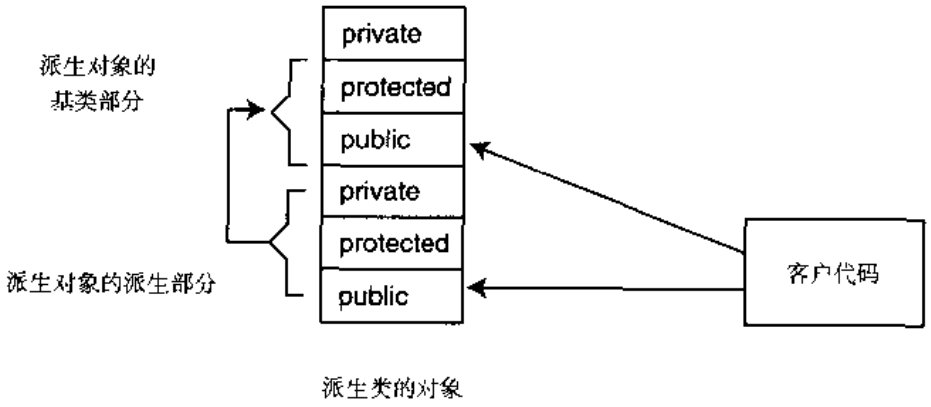


图13-6　当派生模式为公共时，从派生类对象和客户代码访问基类和派生类的服务

我们也可以看到，派生类对象仅仅能访问从基类继承而来的公共和受保护成员。要想访问它自己从基类继承而来的私有成员，派生类方法应使用基类的访问函数。这似乎不太合理，派生类对象居然不能访问它自己的成员！在此之前我们从没有讨论过这种情况。

另一方面，派生类是基类的一个客户。而基类的一些元素，尤其是数据，它们的设计可能随着时间而改变。让这些元素可以为派生类对象所访问，可能也需要修改派生类。派生类通过使用非私有的成员函数进行访问，从而不会受到基类变化的冲击。这和建议使用私有数据成员和公共成员函数的逻辑是一样的。

程序13-5给出了一个较小的抽象例子，它解释了派生类对象与它自己的成员之间的关系。这里的Derived类是以公共模式从Base类中派生而来的。Base类与Derived类一样有public、protected、private成员。Derived类在它的publD()方法中可以访问自己的成员privD和protD（这很显然）。它也可以访问从Base类继承而来的受保护和公共成员：protB和publB()。然而，Derived类访问从Base类继承而来的私有成员privB，就是一个语法错误，即使这个成员的内存分配在Derived对象之内。Client类在它的构造函数中创建了一个Derived类的对象d，并且访问在Derived类中定义的公共服务publD()，以及在基类中定义的公共服务publD()。它不能访问Base类和Derived类的非公共成员。由于我们只想说明对Base类和Derived类功能的访问权限，因此，程序不需输出任何结果，仅仅产生编译时的错误信息。

程序13-5　在公共继承时，从Derived类和客户代码访问一个派生类对象的Base和Derived的成员

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                    // accessed from Base only
   protected: int protB;                  // accessed from Base and Derived
```

```
    public: void publB()                    // access from Base, Derived, Client
    { privB = 0; protB = 0; } } ;           // OK to access its own data

class Derived : public Base {               // public mode of inheritance
    private:  int privD;
    protected: int protD;
    public: void publD()
    { privD = 0;  protD = 0;                // OK to access its own data
      protB = 0;                            // OK to access inherited members
//    privB = 0;                            // no access to inherited members
      } } ;

class Client {
public: Client()                            // Client class constructor
{ Derived d;                                // object of the derived class
  d.publD();                                // OK to access public services
  d.publB();                                // OK to access public Base services
  // d.privD = d.protD = 0;                 // no access to non-public services
  // d.privB=d.protB=0; }                   // no access non-public Base services
   } ;

int main()
{ Client c;                                 // create the object, run the program
  return 0;
  }
```

公共派生是最自然的继承模式，因为它保留了类之间的“是一个”关系。在公共派生里，派生类对象为客户代码提供基类的全部公共特征和它自己增加的公共服务。这样继续继承下去是不受限制的。

注意 对公共派生模式而言，从基类中继承而来的成员在派生类对象中仍然保留它们原有的访问状态（public、protected、private）；而所有在派生类中定义的和从基类中继承而来的公共服务对客户代码都是有效的。这就是继承最自然的模式。

程序13-6显示了一个使用继承的比较大的例子。基类Point提供了两个公共服务set()和get()来访问它的数据成员x和y。派生类VisiblePoint在此基础上增加了数据成员visible和成员函数show()、hide()及retrieve()。方法show()设置数据成员visible的值为1，这样图形包就可以显示这个点。方法hide()设置数据成员visible的值为0，这样就不会显示这个点。继承模式是公共的、可见的和隐藏的点使用枚举类型要比使用字面值更好，但在这里使用字面值是为了使程序代码短小。

程序13-6 在公共继承时，访问一个派生类对象的基类成员

```
#include <iostream>
using namespace std;

class Point {                                   // base class
  int x, y;                                     // private base data
public:
  void set (int xi, int yi)
    { x = xi;  y = yi; }
  void get (int &xp, int &yp) const             // public base methods
    { xp = x;  yp = y; } } ;
```

```
class VisiblePoint : public Point {          // colon: before public
  int visible;

public:                                      // colon: after public
   void show()
     { visible = 1; }

   void hide()
     { visible = 0; }

   void retrieve(int &xp, int &yp, int &vp) const
     { xp = x;  yp = y;                      // syntax error: comment it out!
       get(xp,yp);                           // base public method is accessed
       vp = visible; } } ;                   // derived private data: OK
int main ()
{
  VisiblePoint a,b;  int x,y,z;       // define two derived objects
  a.set(0,0);  b.set(20,40);          // call base public function
  a.hide();  b.show();                // call derived public methods
  a.get(x,y);                         // call base public function
  b.retrieve(x,y,z);                  // call derived public method
  cout << " Point coordinates: x=' << x << "  y=" << y << endl;
  cout << " Point visibility:  visible=" << z << endl;
  return 0;
  }
```

在客户代码中，公共的基类成员函数Point::set()和Point::get()如同VisiblePoint的公共方法一样是可访问的。任何VisiblePoint类的对象都能为它们的客户提供这些服务。

在VisiblePoint类中，私有数据成员Point::x和Point::y是不可访问的；如果试图运行这个程序，在成员函数retrieve()的第一行会有一个语法错误。但有两个补救措施。第一个是将Point类的数据成员定义为protected。假如这些数据成员已在Point类中定义为protected，那么它们可以在类VisiblePoint的retrieve()方法中访问。然而，它们仍不能被客户代码访问。第二个补救措施是在VisiblePoint类的成员函数中使用Point类的访问函数去访问私有的基类数据。在retrieve()方法的第二行我们演示了第二个补救方案。当retrieve()方法的第一行（有语法错误）被注释掉后，程序运行并产生如图13-7的输出结果。

```
Point coordinates: x=20  y=40
Point visibility:  visible=1
```

图13-7　去掉了语法错误的程序13-6的输出结果

第一个补救方案（即定义基类数据为protected而不是private）可能更好一些，因为它只要求基类有很少的访问函数，并且可以简化派生类中的代码。对那些喜欢使用访问函数的人而言，直接从派生类访问受保护的基类数据，就像在客户-服务器关系中直接访问公共数据成员一样，将会破坏封装性。正如我们已经提到的那样，这种看法有些道理。但是问题波及的范围非常非常小。如果发现所涉及的问题很重要，可以让基类数据私有化并在派生类中使用访问函数。否则，将基类数据定义为私有的即可，不用再过多地担心封装的问题。

注意 派生类对象不能访问从基类继承而来的私有成员，即使它们“属于”派生类对象。要想在派生类中访问这些成员，可使用基类部分的访问函数，或者更好一些的方法是在基类中定义这些成员为protected。对派生类而言，受保护的基类成员与公共成员一样是可访问的。对客户代码而言，受保护的基类成员如同私有部分一样，是不可访问的。

13.4.2 受保护继承

受保护继承是限制客户代码访问基类服务的机制。从基类继承的公共和受保护成员都在派生类中变成受保护的。

这些基类部分的服务在进一步的继承中依然有效，并且可以在派生类的方法中使用，但客户代码不能通过派生类对象去访问基类的公共服务，因为它们现在是受保护的。图13-8显示了与受保护继承模式相关的改变，说明了基类中的公共部分变成了受保护的。虚线显示了对派生类对象的这个部分的访问将会被拒绝。

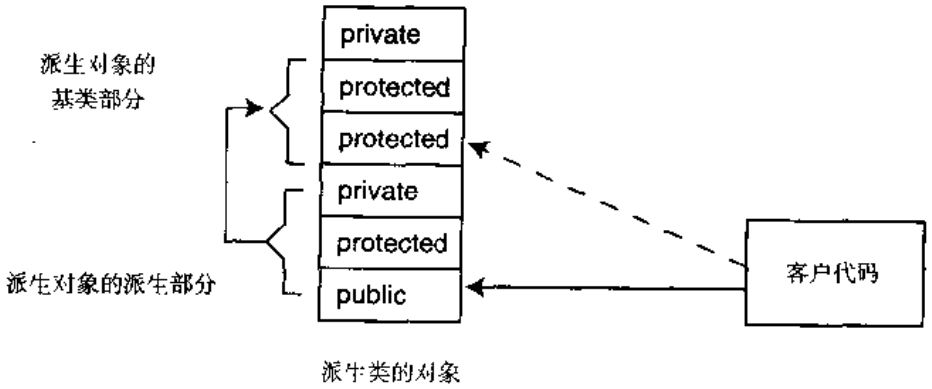


图13-8 当派生模式为受保护时，从一个派生类对象和客户代码中访问基类和派生类的成员

程序13-7是从程序13-5抽取出来的小例子，这里公共模式的继承被受保护模式的继承所代替。这个例子解释了派生类对象和它自己的成员之间的关系，也解释了派生类对象和它的客户代码之间的关系。

程序13-7 当派生模式为受保护时，在Derived类和客户代码中访问Derived类对象的Base类和Derived类成员

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                  // accessed from Base only
   protected: int protB;                // accessed from Base and Derived
   public: void publB()                 // no access from Derived client
   { privB = 0; protB = 0; } } ;        // OK to access its own data

class Derived : protected Base {        // protected inheritance
   private:  int privD;
   protected: int protD;
   public: void publD()
   { privD = 0;  protD = 0;             // OK to access its own data
```

```
      protB = 0;                          // access to inherited members
//    privB = 0;                          // no access its inherited members
       } } ;

class Client {                            // Client code
public:
  Client()
{ Derived d;  Base b;                     // objects of Derived, Base classes
  d.publD();                              // public part of Derived class: OK
//  d.publB();                            // no access to public Base part
//  d.privD = d.protD = 0;                // non-public Derived parts: not OK
//  d.privB=d.protB=0;                    // non-public Base parts: no access
  b.publB(); }                            // Base object: public part is OK
     }

int main()
{ Client c;                               // create the object, run the program
  return 0;
  }
```

d.publB()将派生类对象作为目标对象去调用公共的Base类成员函数publB()，这在前一个版本中（程序13-5）是可行的，然而，在程序13-7中，这是一个语法错误。注意只有当通过Derived类对象访问公共的Base成员时，访问才被拒绝。在Client()缺省构造函数的结尾处，我们用Base类对象b作为目标对象调用了publB()成员函数。这没有问题，一个类的客户可以访问这个类的公共成员，这个服务只是对于Derived类的客户才不可访问。

程序13-8是程序13-6经改进后的例子，这里公共模式的继承被受保护模式的继承代替。基类Point提供相同的公共服务set()和get()，但派生类VisiblePoint的客户不能使用这些服务——它们在VisiblePoint类对象中是受保护的。如果试图在构造函数Client()中这样做将会导致语法错误。要想解决这个问题，我们给VisiblePoint类增加一个新的服务：initialize()，它可代替set()和get()来访问继承的数据成员x和y。注意，现在派生类在访问基类部分的数据时不会有问题，因为我们让基类数据成为受保护的。在派生类成员函数retrieve()中，我们把对基类函数get()的调用注释掉了，以避免不必要的复杂性。

程序13-8　在受保护的继承模式下，访问派生类对象的基类成员

```
#include <iostream>
using namespace std;

class Point {                                   // base class
protected:
  int x, y;                                     // protected base data
public:
  void set (int xi, int yi)
    { x = xi;  y = yi; }
  void get (int &xp, int &yp) const             // public base functions
    { xp = x;  yp = y; } } ;

class VisiblePoint : protected Point {          // protected inheritance
  int visible;

public:
```

```
    void show()
      { visible = 1; }

    void hide()
      { visible = 0; }

    void retrieve(int &xp, int &yp, int &vp) const
      { xp = x;  yp = y;                        // access to protected data is OK
//        get(xp,yp);                           // no need for extra complexity
        vp = visible; }
  void initialize(int xp, int yp, int vp)       // new public service
    { x = xp;  y = yp;                          // access to protected base data
      visible = vp; } } ;                       // access to derived private data

int main ()
{
  VisiblePoint a,b;  int x, y, z;               // define two derived objects
  b.initialize(20,40,1);                        // initialize derived object
// a.set(0,0);  b.set(20,40);                   // now this is a syntax error
  a.hide();  b.show();                          // derived public methods: OK
// a.get(x,y);                                  // and this is a syntax error
  b.retrieve(x,y,z);                            // derived public method: OK
  cout << " Point coordinates: x=" << x << "  y=" << y << endl;
  cout << " Point visibility:  visible=" << z << endl;
  return 0;
  }
```

如果注释掉有语法错误的这两行，程序就可以运行，并产生与程序13-6一样的输出（参见图13-7）。

我们希望大家喜欢使用公共继承而不是受保护继承。公共继承这种技术对基类已经提供的服务添加了新的服务，或者用对派生类的客户代码更有用的方法替换了一些服务（不改变这些服务的名字）。在所有公共继承的例子中，派生类对象和基类对象之间的关系是“是一个”的关系。对客户代码而言，一个存款账户对象“是一个”另外带有支付利息功能的账户对象；一个可见的点对象“是一个”另外带有显示和隐藏功能的点对象。

对于受保护继承而言，情况就完全不一样了，这个技术快速产生一个使用基类的非公共服务（程序13-8中的数据成员x和y）的类，但不给它的客户提供基类的公共服务（程序13-8中的方法set()和get()）。相反，它提供了一组不同的服务（程序13-8中的方法initialize()），出于某种原因，这些服务更适合于客户代码。

在程序13-8中，一个VisiblePoint类的对象不是一个Point对象。Point对象为它的客户提供了方法set()和get()，而VisiblePoint对象没有这样做。

另一个使用受保护继承的流行做法是设计一个堆栈类，以让客户只能从一端访问数据，它从一个（可让客户访问任何元素的）数组类派生而来。使用受保护继承，设计者拒绝让客户使用数组方法。取而代之的是堆栈提供的方法push()和pop()，以便让客户访问堆栈的顶部。

在这一章的开始，我们提到了继承和复合之间的重要不同。派生类为它的客户提供了基类的所有公共服务，复合类不为它的客户提供它的元素服务，除非这些服务被一个复合类的方法所支持。（在该例子里，类Rectangle提供了move()服务。）

如果想从客户代码中去掉一些已有的服务，不要使用继承，用类的复合来代替。

程序13-9显示的是和程序13-8相同的例子，但类VisiblePoint现在有一个Point类的数据成员，而不是以受保护方式从Point继承而来。这个例子的输出与图13-7的一样。

程序13-9　使用类复合来代替继承

```
#include <iostream>
using namespace std;

class Point {                                    // component class
private:
  int x, y;                                      // private data
public:
  void set (int xi, int yi)
    { x = xi;  y = yi; }
  void get (int &xp, int &yp) const              // public method
    { xp = x;  yp = y; } } ;

class VisiblePoint {                             // no inheritance, composition
    Point pt;                                    // private component
    int visible;

public:
   void show()                                   // new service to client
     { visible = 1; }

   void hide()                                   // new service to client
     { visible = 0; }

   void retrieve(int &xp, int &yp, int &vp) const   // replace
     { pt.get(xp,yp);                            // services are hidden from client
       vp = visible; }

   void initialize(int xp, int yp, int vp)       // replace
    { pt.set(xp,yp);                             // services are hidden from client
      visible = vp; } } ;                        // just like private data are hidden

int main ()
{
VisiblePoint b;  int x, y, z;                      // define an aggregate object
b.initialize(20,40,1);                             // aggregate service
b.show();                                          // aggregate service
b.retrieve(x,y,z);                                 // aggregate service
cout << " Point coordinates: x=" << x << "  y=" << y << endl;
cout << " Point visibility:  visible=" << z << endl;
return 0;
}
```

因为使用了复合类，基类Point的服务set()和get()可以从客户代码中去掉。Point的数据成员隐藏在VisiblePoint对象里，不能供客户代码使用。因此，客户代码不能对一个VisiblePoint对象做一些像对Point对象做的操作：例如在屏幕上移动一个点，而不管它是否可见。VisiblePoint类为其客户代码提供了自己的接口：成员函数initialize()和retrieve()，它们要求客户代码处理点的可见性。

当设计一个新类为客户代码服务，而且存在着一个可以重用其设计的类时，这个例子中的思想也同样适用。假如客户不仅需要这个已有类的所有服务，还需要其他服务时，可以以

公共模式继承这个类。客户代码将使用派生类对象所继承的服务和增加的服务。假如是新类而不是客户代码要使用已有类的服务，则不要使用继承，也不要让它成为受保护的，使用类复合就可以了（有关复合的详细情况请参阅第12章）。

当要设计一个类（或一些类）为客户服务，并且想通过继承以模块的方式逐渐构造这个类时，受保护的继承将会很有用。

例如，客户代码需要类D1，我们想从类D中派生出D1，而类D又是从类B中派生而来的。如果以受保护继承模式从类B派生类D，再从类D派生类D1，我们就可以创建具有类D的所有公共和受保护类型的服务的类D1。这些服务也包括基类B的所有公共和受保护类型的服务。D1的客户代码不会用类D和类B的公共服务，因为使用了受保护的继承模式，这些服务在客户代码被剥夺了。换句话说，受保护继承是一个限制客户代码访问基类的公共服务的方法，但该方法不限制从派生类中访问这些服务，对进一步继承也没有限制。

注释 受保护继承模式从使用派生类对象的客户代码中去掉了继承来的基类公共服务。这曲解了“是一个”关系。如果这个关系不重要，可以使用类复合来代替受保护继承。如果想使用继承，则应采用公共继承。（这只是个人意见）

13.4.3 私有继承

私有继承是限制访问基类服务的技术，它不仅可以对派生类的客户有限制，而且也对派生类的派生类有限制。

当基类作为一个私有基类时，所有公共和受保护的基类成员在派生类中都变成了私有成员，它们不能被派生类的客户访问，也不能被从派生类继承来的方法访问。它们仅在必要时由派生类的方法访问。

将基类作为私有模式派生和作为其他模式派生有很重要的不同。使用受保护或公共继承，访问规则是可传递的。如果派生类作为进一步派生（受保护或公共模式）的基类，顺着继承层次往下的派生类，拥有与直接从基类派生的派生类一样对基类成员的访问权限。然而，假如是私有派生模式，而且派生类用于进一步的派生，它的后裔不能访问基类的任何成员。基类的受保护和公共成员只能被基类的直接派生类访问。这阻止了派生类的设计者在进一步的继承中使用基类的服务。

和受保护继承一样，基类的公共接口（数据成员和成员函数）不再是派生类接口的一部分——它对客户而言是私有的。

这些关系如图13-9所示。如果继承的模式是私有的，从基类派生来的派生类对象（这还不是一个很高的继承层次）对自己从基类继承而来的元素没有访问权限。

程序13-10再次显示了一个短小而抽象的例子，这次我们使用了私有继承。就派生类对象对它的基类部分的访问权限而言，与前一个例子（使用受保护继承模式）是相同的。在这个例子中，我们还引入了另一个继承Derived的类Derived1。这个继承关系使用的是公共继承，但这无关紧要。我们将看到的是从Base类私有继承而来的效果。

我们注释掉了那些有危害性的语句，以使代码能通过编译。可以看到派生类能访问所有非私有的基类成员，这不依赖于继承的模式。类似地，类Derived1能访问它自己的“基类”（Derived类）的所有非私有成员，这也不依赖于继承的模式。然而，类Derived1不能访问

Base类的任何成员，因为它的基类（Derived类）是从Base类中以私有模式继承而来的。就客户代码而言，私有继承与受保护继承是相似的：它们使客户代码不能访问派生类对象的所有基类元素，包括公共成员。

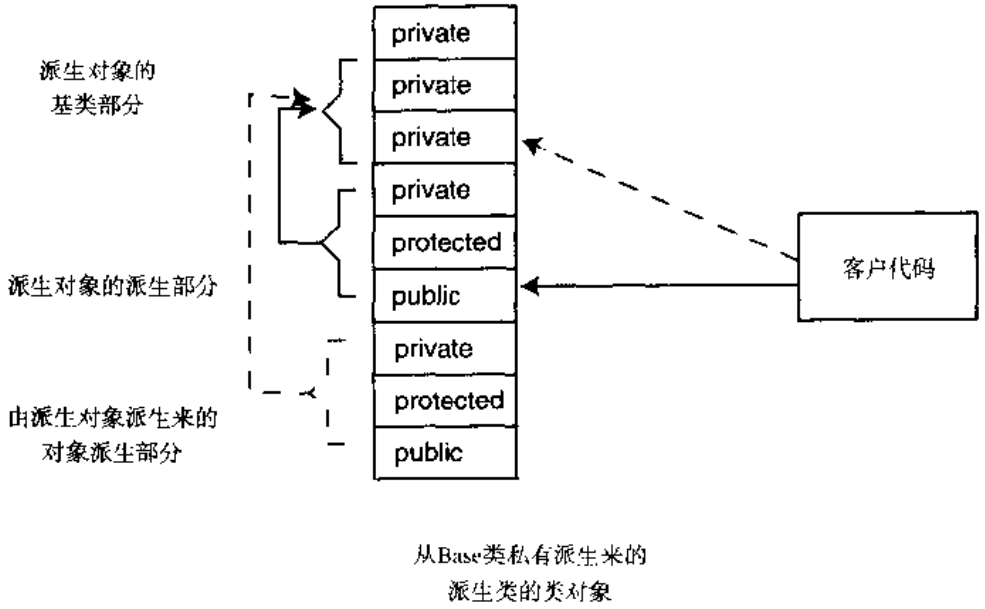


图13-9　私有继承模式下，访问一个派生类对象的基类成员

程序13-10　当Derived类以私有模式继承Base类时，在继承层次中访问Base成员

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                  // accessed from Base only
   protected: int protB;                // accessed from Base and Derived
   public: void publB()                 // accessed from Base and Derived
   { privB = 0; protB = 0; } } ;        // OK to access its own data

class Derived : private Base {          // private inheritance
   private:  int privD;
   protected: int protD;
   public: void publD()
   { privD = 0;  protD = 0;             // OK to access its own data
     protB = 0;                         // OK to access inherited members
 //  privB = 0;                         // not OK to access its inherited members
                                              } } ;

class Derivedl : public Derived {       // class derived from derived
public: void publDD()
{ // privD = 0;                         // no access to private "base" data
  protD = 0;                            // OK to access protected "base" data
  publD();                              // OK to access public "base" data
//  protB = 0;                          // no access to any part of "private base"
//  publB();                            // no access to any part of "private base"
} } ;

class Client {
```

```
public:
  Client()
{ Derived d;  Base b;                    // objects of derived and base classes
  d.publD();                             // public part of Derived class: OK
//  d.publB();                           // public Base part of Derived: not OK
//  d.privD = d.protD = 0;               // non-public part of Derived: no
//  d.privB=d.protB=0;                   // non-public Base part of Derived: no
  b.publB(); }                           // public Base part of Base object: OK
   }

int main()
{ Client c;                              // create the object, run the program
  return 0;
  }
```

私有继承允许我们通过重用编写新的服务。但这不是子类型（subtype）关系：如果从类Array私有地派生了Stack类，一个Stack对象不是一个Array对象，它不对Stack类的客户或者从Stack类进一步派生的派生类提供Array类的服务。一个Stack对象可以有一个Array对象作为它的元素。使用私有或受保护的继承不是一个好的设计，更好的方法是使用类的复合。

然而，一些专家却认为这种继承模式是有用的，因为它迫使派生类使用基类的访问方法去访问私有数据成员，就像其他类的其他客户一样。正如我们已经指出的，这是个有争议的问题。

多态性（将在下一章讨论）只有在公共继承下才有效，这是使用公共继承的另一个理由。应避免使用受保护和私有继承。

13.4.4 调整对派生类中基类成员的访问

C++允许派生类的程序员避免受到受保护继承和私有继承规则的访问限制。如何避免呢？将基类成员在基类对象中的访问权限显式地返回给派生类对象中的基类成员。

程序13-11中的骨架例子再次从Base类私有派生Derived类。在Derived类的定义中，我们恢复了数据成员Base::protB的受保护状态，同时也恢复了成员函数Base::publB()的公共状态。注意无论是数据成员还是成员函数，其语法都相同。这样并不改变Derived类的访问权限，对于任意模式的继承，Derived类都可以访问其基类中的所有非私有成员。但这样改变了Derived1类的访问权限，与Derived类相似，它也可以访问其基类中的所有非私有成员。客户代码也发生了改变，现在能访问Base::publB()，就好像Derived类是公共继承而不是私有继承Base类一样。

程序13-11 调整对Derived类中的Base成员的访问权限（私有继承）

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                   // accessed from Base only
   protected: int protB;                 // accessed from Base and Derived
   public: void publB()                  // accessed from Base and Derived
   { privB = 0; protB = 0; } } ;         // OK to access its own data

class Derived : private Base {           // private inheritance
```

```
    private:  int privD;
    protected: int protD;
    protected:
      Base::protB;                          // available for further derivation
  public:
    Base::publB;                            // available for client access
    public: void publD()
    { privD = 0;  protD = 0;                // OK to access its own data
      protB = 0;                            // OK to access its inherited members
//    privB = 0;                            // private inherited member: no access
    } } ;

class Derived1 : public Derived {           // class derived from derived
public: void publDD()
{ // privD = 0;                             // no access to private "base" data
  protD = 0;                                // OK to access protected "base" data
  publD();                                  // OK to access public "base" data
  publB();                                  // OK if it is made public in Derived
  protB = 0;                                // OK if it is made protected in Derived
    } } ;

class Client {
public: Client()
{ Derived d;  Base b;                       // objects of derived and base classes
  d.publD();                                // public part of Derived class: OK
  d.publB();                                // OK if it is made public in Derived
//  d.privD = d.protD = 0;                  // non-public part of Derived: no
//  d.privB=d.protB=0;                      // non-public Base part of Derived: no
  b.publB();                                // public Base part of Base object: OK
   } } ;

int main()
{ Client c;                                 // create the object, run the program
  return 0;
  }
```

我们可以看到，在使用私有继承时，如果关闭对某些元素的访问权限同时开放对另外一些元素的访问权限，就会让设计十分费解。将设计变得如此难懂，以至于其他程序员要猜测程序的用处。

实际上，C++不仅允许调整访问权限，同样也可以在派生类中将其改变得与基类中不同。我们惟一不能做的是，在派生类中通过将私有的基类成员设置为受保护的或公共的而将它们改成非私有的。

13.4.5 缺省继承模式

在定义继承时，最好是显式地明确地指定继承模式，但C++也允许使用缺省的继承模式。这时，我们假设客户端代码程序员和维护人员有足够的知识理解我们的意思，即使我们没有确切地说明。

一个派生类的缺省继承模式为私有继承。如果忘记显式地指定继承模式，编译程序将认为使用的是私有模式的继承。在程序13-12的骨架程序中，我们忘记说明自己的意图。结果，客户代码不能用Derived类目标访问从Base类继承而来的公共方法publB()。

程序13-12 使用缺省继承模式的派生类的例子

```
class Base {
  private: int privB;                  // accessed from Base only
  protected: int protB;                // accessed from Base and Derived
  public: void publB()                 // accessed from Base and Derived
  { privB = 0; protB = 0; } } ;        // OK to access its own data

class Derived : Base {                 // it is private by default
  private:  int privD;
  protected: int protD;
  public: void publD()
  { privD = 0;  protD = 0; protB = 0; } } ;      // OK to access

int main()
{ Derived d;                           // object of the derived class
  d.publD();                           // OK to access public part of Derived class
//  d.publB();                         // not OK to access public part of Base class
  return 0;
  }
```

实际上，并不像看上去那么简单。只是对于使用class关键字定义的派生类，继承的缺省模式才为private。而C++的结构的缺省继承模式为public。除了对数据成员和成员函数的缺省访问权限之外，class和struct关键字表示同样的意义。对于class，缺省的访问权限是private；对于struct，缺省的访问权限是public。除此之外，它们的意义是相同的。在结构中可以有成员函数，也可以重载这些成员函数，并为它们设置缺省参数，还可以有构造函数和析构函数、其他类（与结构）的数据成员、成员初始化列表等其他所有区别面向对象程序设计和过程程序设计的元素。当然，可以继承一个结构，也可以从一个类或结构派生一个结构，所有这些在C++中都是合法的。对于用struct关键字定义的派生类，缺省的继承模式为public而不是private。

程序13-13是一个使用关键字struct定义的Derived类的例子，由于Derived类是在缺省继承模式下从基类派生而来，因而继承模式是公共的。

程序13-13 使用缺省继承模式派生结构的例子

```
class Base {
   private: int privB;                 // accessed from Base only
   protected: int protB;               // accessed from Base and Derived
   public: void publB()                // accessed from Base and Derived
   { privB = 0; protB = 0; } } ;       // OK to access its own data

struct Derived : Base {                // it is public by default
   private:  int privD;
   protected: int protD;
   public: void publD()
   { privD = 0;  protD = 0; protB = 0; } } ;      // OK to access

int main()
{ Derived d;                           // object of the derived class
  d.publD();                           // OK to access public part of Derived class
  d.publB();                           // Hey, this is perfectly legitimate now!
  return 0;
  }
```

请不要认为C++这样处理只是为了让程序员更加糊涂。这与C++中的访问类成员的缺省规则是完全一致的。当使用class关键字定义一个类时，对于这个类的成员的缺省访问权限是private。当使用struct关键字定义一个类时，对于这个类的成员的缺省访问权限是public。

类似地，当使用class关键字从一个类派生另一个类时，缺省继承模式为private。当使用struct关键字从一个类派生另一个类时，缺省继承模式为public。其区别完全在于派生类的定义方式上。可以使用class关键字或struct关键字定义基类，这不会影响派生类的继承模式。

依赖缺省因素并不是一个好的做法。

13.5　在继承下的作用域规则和名字解析

在C++中，派生关系下的类作用域可被看作是嵌套的。按照这个观点，派生类的作用域包含在它的基类作用域之内。

根据嵌套作用域的一般规则，内部作用域中的任何定义对于外部作用域乃至全局作用域是不可见的。反过来，外部作用域中的任何定义对于内部作用域是可见的。在下面的例子中，变量x是在外部函数作用域中定义的，变量y是在内部语句块作用域中定义的。在内部作用域中访问变量x是合适的，但在外部作用域中访问变量y是无效的。

```
void foo()
{ int x;              // outer scope: equivalent to base class
   { int y;           // inner scope: equivalent to derived class
     x = 0; }         // ok to access the name from outer scope
  y = 0; }            // syntax error: inner scope is invisible outside
```

在这个例子中，外部作用域扮演基类及其成员的角色，内部作用域扮演派生类及其成员的角色。可以从派生类中访问基类的成员，但不能从基类访问派生类的成员。

这意味着派生类的成员在基类作用域中是不可见的。这和我们的直觉是一致的，因为基类应该在派生类编写之前被设计、实现和编译。因此很自然地，基类的成员函数不能访问派生类的数据成员或成员函数。

相反，基类成员在外部作用域中，因此对于派生类方法是可见的。这也和我们的直觉相吻合，因为派生类对象“是一个”基类对象，它拥有基类所有的数据成员和成员函数。按照这个观点，基类和派生类之间关系的作用域模型并不很有用，因为它并没有超越我们的直觉。但当派生类和基类使用相同的名字时这个模型非常有用。不同的语言使用不同的规则解析名字冲突，因此被C++采纳的这种嵌套作用域模型就有助于我们培养编写C++代码的直觉。

派生类的作用域嵌套在基类的作用域之中，这意味着在派生类中，派生类的名字隐藏了基类名字。类似地，在派生类的客户代码中，派生类的名字也会隐藏基类的名字。这是一条非常重要的规则，应该成为我们的编程直觉的一部分：如果派生类和基类使用了相同的名字，基类的名字将没有任何机会，因为使用的将是派生类名字的意义。

下面我们将通过例子澄清一下这条规则。如果在派生类的成员函数中发现了一个没有作用域运算符的名字，编译程序会试图将这个名字解析为该成员函数的局部名字。在下面的代码例子中，有四个变量的名字都为x。所有这些变量都是同一类型，但这并不重要，它们的类型也可以不同，或者其中某些名字也可以标识函数。我们正在讨论的通用规则都同样适用。

```
int x;                              // outer scope: can be hidden by class or function
class Base {
  protected: int x;                 // base name hides global names
} ;

class Derived : public Base {
  int x;                            // derived name hides base names
public:
  void foo()
  { int x;
    x = 0; } } ;                    // local variable hides all other names

class Client {
public:
  Client()
  { Derived d;
    d.foo(); } } ;                  // using object d as a target message

int main()
{ Client c;                         // define the object, run the program
  return 0;  }
```

在这段程序中，Derived类的成员函数foo()中有一个局部变量x，Derived类有一个名为x的数据成员，Base类中也有一个名为x的数据成员，文件作用域中还有一个全局变量x。Derived::foo()中的x=0;语句将局部变量x设置为0。派生类数据成员Derived::x、基类数据成员Base::x及全局变量x都被这个局部变量给隐藏了，因为这个局部变量定义嵌套在最内层的作用域中。

我们将foo()方法中变量x的定义注释掉，x=0;语句将不能解析为局部变量，因为找不到这个局部变量名。如果这个变量名在语句的作用域中（这里是指派生类的成员函数）找不到，编译程序将根据这个名字的引用语法查找派生类作用域中的数据成员和成员函数。在上面的例子中，如果没有了Derived::foo()中的局部变量x，则派生数据成员Derived::x将被派生成员函数Derived::foo()中的x=0;语句设置为0。

如果在类作用域中仍找不到成员函数中所用到的名字，编译程序将搜索其基类（如果在基类中找不到，则搜索基类的祖先类）。搜索中发现的第一个名字用来产生目标代码。在上面的例子中，如果Derived类中的两个x变量（局部变量和数据成员）都不存在，则是基类的数据成员Base::x被x=0;语句设置为0。

最后，如果在任何基类中都没有找到这个名字，编译程序将在文件作用域的声明中搜索（在文件作用域中定义为全局对象，或在其他地方定义而在该文件作用域中声明为extern的全局对象）。如果在此过程中找到了则使用；如果找不到则产生语法错误。在上面的例子中，如果Derived类和Base类都没有使用x，则全局变量x被Derived::foo()中的x=0;语句设置为0。

类似地，如果派生类的客户给派生类的对象发送消息，编译程序将首先查找派生类，如果没有找到，则再向上查找基类的定义（或基类的祖先类的定义）。如果派生类和其中一个基类使用了相同的名字，则使用派生类中的解释。如果编译程序在派生类中找到了所要的名字，就不会向上查找基类，派生类中的名字隐藏了基类中相应的名字，基类的名字没有任何机会被发现。

对Base类和Derived类修改后的例子如下所示。这个例子中有两个foo()函数：一个

是Base类的public成员函数，另一个是Derived类的public成员函数。与前面的例子相似，客户代码定义了一个Derived类的对象，并发送foo()消息给这个对象。既然Derived类定义了成员函数foo()，就调用派生类的成员函数。如果Derived类没有定义foo()函数，编译程序就会产生对Base类的foo()函数的调用。只有当Derived类中不使用相同的名字时，Base类的函数才有机会被调用。

```
class Base {
  protected: int x;
public:
  void foo()              // Base name is hidden by the Derived name
  { x = 0; } } ;

class Derived : public Base {
public:
  void foo()              // Derived name hides the Base name
  { x = 0; } } ;

class Client {
public:
  Client()
  { Derived d;
     d.foo(); } } ;       // call to the Derived member function

int main()
{ Client c;               // create an object, call its constructor
  return 0;  }
```

注意，这个例子中没有涉及全局作用域。如果Derived类和Base类（或任意祖先类）中都没有foo()成员函数，则d.foo()函数调用是一个语法错误。如果函数foo()定义在全局作用域中，也不能以d.foo()方式来调用这个函数。

```
void foo()
{ int x = 0; }
```

这个全局函数不会被Derived（或Base）类中的foo()成员函数所隐藏，因为它有不同的界面。成员函数需要用目标对象来调用，而调用全局函数时只需要使用函数名即可。

```
foo();                    // call to a global function
```

我们所讨论的函数调用有不同的语法形式：

```
d.foo();                  // call to a member function
```

不能用这种语法形式调用全局函数，因为它包括了目标对象，只能调用类的成员函数。

13.5.1　名字重载与名字隐藏

注意在前面的例子中，没有考虑参数个数及相应的类型。这并不是疏忽，而是因为它们不是影响因素。

这样说当然是开玩笑，它们其实很重要。当编译程序决定实际参数是否与函数的形式参数相匹配时，函数标识的确很重要。只是它对于嵌套继承作用域的解析并不重要。如果在派生类中找到了所要找的名字，编译程序将停止对继承链的搜索。如果在派生类中所找到的函数从参数匹配的角度来看不是很合适，会怎么样呢？太糟糕了，会出现语法错误。如果基类

有一个更好的匹配，即函数名字相同，函数调用与函数标识也完全匹配，那会发生什么事情呢？也很糟糕，因为太迟了。基类函数根本没有机会被调用。

不幸的是，这与许多程序员的直觉相抵触。请使用例子实际应用这些嵌套规则，以确保锻炼了自己的直觉。下面的例子来自实践。我们已经删除了与嵌套作用域的隐藏无关的内容，只留下了一小部分代码。

程序13-14表示了账户类层次的简化部分。我们只使用了Account类和CheckingAccount类。派生类重定义了基类的成员函数withdraw()，但这不是讨论的重点。客户代码定义了CheckingAccount类的对象，并发送一些消息给这些对象。所发送的消息有的属于基类（getBal()和deposit()），有的属于派生类本身（withdraw ()），这些都很正常。程序的输出结果如图13-10所示。

程序13-14 Account类的继承层次例子

```
#include <iostream>
using namespace std;

class Account {                                  // base class
protected:
  double balance;

public:
  Account(double initBalance = 0)
  { balance = initBalance; }

  double getBal()                                // inherited without change
  { return balance; }
  void withdraw(double amount)                       // overwritten in derived class
  { if (balance > amount)
      balance -= amount; }

  void deposit(double amount)                        // inherited without change
  { balance += amount; }
} ;

class CheckingAccount : public Account {           // derived class
  double fee;

public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }

  void withdraw(double amount)                       // it hides base class method
  { if (balance > amount)
      balance = balance - amount - fee; }
} ;

int main()
{
  CheckingAccount a1(1000);                          // derived class object
  a1.withdraw(100);                                  // derived class method
  a1.deposit(200);                                   // base class method
  cout << " Ending balances\n";
  cout << "   checking account object: " << a1.getBal() <<endl;
```

```
return 0;
}
```

```
Ending balances
  checking account object: 1099.8
```

图13-10　程序13-14的输出结果

尽管这里的客户代码不是很长，但在实际中大约有200页左右。这个程序可改进以反映商务环境的变化。其中一个所需的变化是给CheckingAccount类增加另一个deposit()函数，用于国际电报传输。在这些传输中，交易的费用应根据交易额和交易源而定。这个费用可由客户代码进行计算，并作为参数传送给CheckingAccount类。因此，要支持这个变化，简单的办法是编写另一个带有两个参数的deposit()函数。

```
void CheckingAccount::deposit(double amount, double fee)
 { balance = balance + amount - fee; }
```

处理国际传输和计算所需费用的客户代码只需要对程序添加几页纸。以下是新客户代码调用这个新编写的deposit()函数的例子。

```
a1.deposit(200,5);                    // derived class method
```

到此为止，一切运行正常，改变没有问题，新的代码运行得很好。但是在系统集成时出现了问题。在修改前运行得很正常的200页客户代码不能正常运行了。确切地说，代码完全不能工作，甚至不能通过编译。

请相信，我们在使用C++之前使用了许多语言，但是从来没有见过类似的事情。假设其他程序员在使用C++之前也使用了许多语言，但是也从来没有见过类似的事情。这就是我们需要注意的C++对软件工程的另一贡献。

当然我们都遇到过这样的情况，添加的一些新代码破坏了已有的代码，从而不能再正确运行。通常，这是由于新加入的代码干扰了已有代码所依赖的数据而引起的。但是现有的代码经常可以通过编译。在传统语言中，当添加新代码后，不会在已有代码中得到语法错误。

在C++中，程序由相互链接的类组成，这些类不仅通过数据而且也通过继承相互链接。当然，如果新加入的代码不正确地处理数据，将会使已存在的代码出现语义错误。这在任何语言中都可能出现。但新加入的代码还可以通过继承链使已存在的代码出现语义错误。这种情况只有在C++中才可能出现。这也是为什么我们强调编程直觉、需要了解规则并培养对正确和不正确的C++代码的感觉。

下面我们来分析一下这种“革新”程序出现问题的原因。这里是新修改后的CheckingAccount类：

```
class CheckingAccount : public Account {
  double fee;
public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }
  void withdraw(double amount)                    // it hides base class method
  { if (balance > amount)
      balance = balance - amount - fee; }
```

```
    void deposit(double amount, double fee)          // new method
    { balance = balance + amount - fee; }            // hides base method
} ;
```

当编译处理现有的200页客户代码时，对成员函数deposit()的调用是指向带有一个参数的基类成员函数Account: :deposit()。

```
a1.deposit(200);                          // base class method?
```

根据我们刚讨论的名字解析规则，编译程序分析消息目标对象的类型，发现a1对象属于CheckingAccount类，再查找CheckingAccount类中名字为deposit()的成员函数。当编译程序找到函数后，则停止对继承链的搜索。下一步当然是标识匹配。编译程序发现在派生类中找到的CheckingAccount::deposit()方法有两个参数，而（想调用基类方法的）客户代码只提供了一个参数，编译程序将十分肯定地提示有语法错误。

我们很清楚代码是正确的，因此猜测这可能是编译程序的另一个程序错误。（编译程序到底是什么无关紧要。在学习一种新的语言时，我们常常会认为编译程序有一些程序错误，直到我们更好地理解这种语言为止。）

如果编译程序能够将这种情况当做名字重载来对待，我们就会轻松很多。在基类中已存在一个带有一个参数的deposit()函数，而在派生类中又增加了一个带有两个参数的deposit()函数。但派生类的对象也是基类的对象！它也有一个继承来的带一个参数的deposit()函数。我们的直觉是在派生类中有两个deposit()函数：其中一个带有一个参数，另一个带有两个参数。如果编译程序使用函数名重载规则，挑选了仅带有一个参数的正确函数，事情就好办了。但正如我们在前面讨论的那样，基类的方法被派生类的方法隐藏，基类方法没有任何机会。重载适用于同一作用域中的几个函数。在嵌套作用域的函数中使用的是隐藏。最后，我们放弃这种做法并决定改变自己的想法。这的确需要时间，但是相信大家都能够做到。

警告 C++只在同一作用域中支持函数名重载。在独立的作用域中，函数名不会发生冲突，可以使用相同的函数名，其标识可以相同也可以不同。在嵌套作用域中，嵌套作用域中的名字将隐藏外部作用域中的名字，无论函数标识是否相同。如果类之间存在继承关系，派生类的函数名将隐藏基类中的同名函数；同样，函数标识并不重要。

图13-11显示的是一个含有两个函数的派生类对象，其中一个函数来自于基类，另一个来自于派生类。从客户代码出发的垂直箭头表明编译程序开始在派生类中查找，一旦找到了同名的函数（无论其标识是否相同），编译程序就停止查找，而不会使用名字重载规则去查找基类。如果觉得继承中嵌套作用域的概念很抽象，可以通过这个图来提醒自己查找在第一次匹配时就会停止。

13.5.2 派生类所隐藏的基类方法的调用

对于上述问题有几种补救措施，其中一种办法是在客户代码中指明所要调用的函数。使用作用域运算符可以很好地实现此办法。

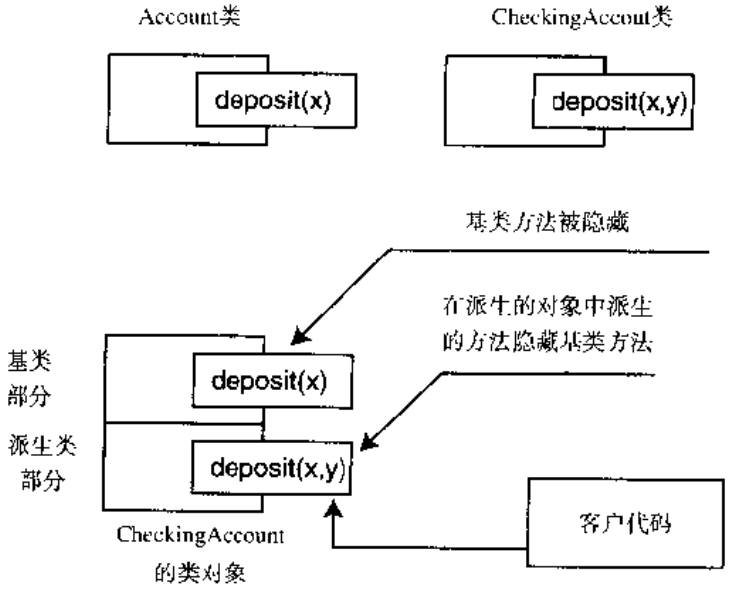


图13-11　在派生类对象中的派生类方法隐藏了基类方法

```
int main()
{ CheckingAccount a1(1000);                    // derived class object
  a1.withdraw(100);                            // derived class method
//  a1.deposit(200);                           // syntax error
  a1.Account::deposit(200);                    // solution to the problem
  cout << " Ending balances\n";
  cout << "   checking account object: " << a1.getBal() <<endl;
  return 0; }
```

不过，不要高兴太早。这种办法有个明显的不足是要求改变现有的代码。面向对象方法的优点是增加代码而不是修改已存在的代码。这种办法不仅要花费大量的精力，而且容易出错。使用这种解决方法等于是自找麻烦。

从软件工程的角度而言，这种方法与我们前面讨论的编写C++程序的基本原则相抵触。哪一个原则呢？我们来看看这个例子中，是谁负责做解决方案中的工作呢？是客户代码。那又是谁负责根据写代码的原则执行这种解决方案呢？是服务器代码。这种解决方案就没有能够将任务推向服务器类。相反，它把任务带到了客户代码中。我们需要确保调用的是基类的方法，即需要显式地声明应该调用基类的函数。这是个残忍的强制性解决方法。

在我们的工作中，一定要使用将任务推向服务器类实现的原则。这指明了我们寻求好的解决方法的方向。让我们分析一下Account类的继承层次。我们的目标应是为这些类增加某个方法（或某些方法）来解决问题。为什么要增加方法？因为我们不想修改已有的代码。为什么我们要把方法增加到继承层次中？因为这些类是客户代码的服务对象，我们要将任务转移到服务器类。

其中一种补救措施是在基类中重载deposit()方法，而不是在派生类中重载。由于两个同名函数属于同一个类，因此位于同一个作用域中，这样就属于合法的C++函数名重载的范畴。这两个类都被派生类所继承，可将派生类对象作为消息的目标对象来调用这两个函数。下面的例子演示了这种方法。

```
class Account {                                    // base class
```

```
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
  double getBal()                                    // inherited without change
  { return balance; }
  void withdraw(double amount)                       // overwritten in derived class
  { if (balance > amount)
      balance -= amount; }
  void deposit(double amount)                        // inherited without change
  { balance += amount; }
  void deposit(double amount,double fee)             // overloads deposit()
  { balance = balance + amount - fee; } } ;

class CheckingAccount : public Account {             // derived class
  double fee;
public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }
  void withdraw(double amount)                       // hides the base class method
  { if (balance > amount)
      balance = balance - amount - fee; } } ;

int main()
{ CheckingAccount a1(1000);                          // derived class object
  a1.withdraw(100);                                  // derived class method
  a1.deposit(200);                                   // existing client code
  a1.deposit(200,5);                                 // new client code
  cout << " Ending balances\n";
  cout << "   checking account object: " << a1.getBal() << endl;
  return 0; }
```

注意这种方法的修改形式是在服务器类中增加代码，而不必修改客户代码；这就将任务推到了Account类，这样做很好。但这种解决方法要求开放并修改基类而不是派生类。这对于控制而言，不是所希望的。在继承层次中，如果类层次越高，就越不能修改这个类，因为修改这个类后会影响其他的派生类；当类层次越低，开放并修改这个类也就越安全。

这种方法的另一个问题是，作用域规则只允许基类的成员函数访问基类数据成员，而不能访问派生类的数据成员。在本例中，这不是问题，两个deposit()方法都只需要基类的数据成员。但其他情况不会都这样。新的方法可能需要在派生类中定义而在基类中没有的数据。例如，存款交易可能要收取标准的取款费用。那么，新的方法deposit()只能在派生类中实现。

```
void CheckingAccount::deposit(double amount, double fee)
  { balance = balance + amount - fee - CheckingAccount::fee; }
```

但是，将这个新的方法增加到派生类中后，又出现了嵌套名字作用域的问题：这个函数隐藏了基类的deposit()函数，使得现有的调用带有一个参数的deposit()的代码出现语法错误。

解决这个问题的一个较好办法是将新的deposit()放到它所属的派生类中，为了使现有的对带有一个参数的deposit()函数的调用合法，可以在派生类中重载deposit()方法，而不是在基类中重载。而且，派生类是客户代码的服务器类，这种方法将任务推到了服

务器类。

提示　在编写C++代码时，要经常争取将职责从客户代码推向到服务器代码。这样，客户代码只需表达计算的意义，而不是计算的细节。这是一条非常普遍的规则，会带来不少好处。

程序13-15演示了这种方法。派生类有两个成员函数deposit()，它们的参数个数以及相应的类型不同。由于它们属于同一个类，因而支持名字重载规则。这样，新加入的代码和已有的代码可以使用不同的参数调用派生类的成员函数。带有一个参数的成员函数应该做的所有事情是调用基类的同名成员函数（将任务推给服务器类）。程序的运行结果如图13-12所示。

程序13-15　Account类的继承层次例子

```
#include <iostream>
using namespace std;

class Account {                                    // base class
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
  double getBal()                                  // inherited without change
  { return balance; }
  void withdraw(double amount)                     // overwritten in derived class
  { if (balance > amount)
      balance -= amount; }
  void deposit(double amount)                      // inherited without change
  { balance += amount; }
} ;

class CheckingAccount : public Account {           // derived class
  double fee;

public:
  CheckingAccount(double initBalance)
  { balance = initBalance;  fee = 0.2; }

  void withdraw(double amount)
  { if (balance > amount)
      balance = balance - amount - fee; }

  void deposit(double amount)                      // hides the base class method
  { Account::deposit(amount); }                    // call to a base function

  void deposit(double amount, double fee)          // hides base method
  { balance = balance + amount - fee - CheckingAccount::fee; }
} ;

int main()
{
  CheckingAccount a1(1000);                        // derived class object
  a1.withdraw(100);                                // derived class method
  a1.deposit(200);                                 // existing client code
  a1.deposit(200,5);                               // new client code
```

```
cout << " Ending balances\n";
cout << "   checking account object: " << a1.getBal() <<endl;
return 0;
}
```

```
Ending balances
  checking account object: 1294.6
```

图13-12 程序13-15的输出结果

千万不要被本例中使用的作用域运算符吓倒。CheckingAccount类中带一个参数的deposit()函数也可以写成以下这种形式：

```
void CheckingAccount::deposit(double amount)      // hides base method
  { deposit(amount); }                            // infinite recursive call
```

当编译程序处理这个函数体时，首先在这个函数内查找与deposit()相匹配的名字，如果没有找到，再在类成员中进行查找。直到它找到了CheckingAccount::deposit()并执行调用。结果，这个调用被解释为无穷的递归调用。

程序13-15中的作用域运算符指示编译程序调用基类函数Account::deposit()，从而避免了这个递归调用的陷阱。注意，处理类继承层次的任务以及决定deposit()函数所属类的工作被推向服务器类，而不像第一种弥补方法那样上推给了客户代码。

CheckingAccount类中带有两个参数的depcsit()函数也可以写成如下形式：

```
void CheckingAccount::deposit(double amount, double fee)
  { balance = balance + amount - fee - fee; }
```

当编译程序处理这个函数体时，在函数内查找与fee相匹配的名字，这个名字是函数的第二个参数。尽管CheckingAccount类有一个数据成员为fee，但这个数据成员被函数的参数名隐藏。要访问类的数据成员fee，程序13-15中必须使用重定义作用域规则的作用域运算符。

13.5.3 使用继承改进程序

处理这种程序改进的一个好方法是，避免将问题及其弥补方法混在一起。我们遇到的困难的起源是，在关于国际电报传输的程序13-14和程序13-15中，我们想修改已有的代码（Account类和CheckingAccount类）来适应新环境的要求。

传统的程序设计方法自然会想到这种修改已有的代码的做法。但C++支持的面向对象的程序设计方法让我们有机会采用别的做法。我们可以通过继承已有的类来满足新的要求，而不用想方法修改已有的代码。

没错，我们谈论的正是考虑编写代码的新方法。使用继承意味着要编写新的代码，而不是修改已有的代码。每个试图修改已有代码的人都会知道这两种方法完全不同。C++提供了一个新的办法来解决这个小小的国际电报传输问题：保留已有的200页代码，保留Account类和CheckingAccount类不被修改，引入另一个新的派生类来支持新的客户代码。

```
class InternationalAccount : public CheckingAccount {      // great!
```

```
public:
  InternationalAccount(double initBalance)
  { balance = initBalance; }
  void deposit(double amount, double fee)                    // hides base method
  { balance = balance + amount - fee - CheckingAccount::fee; }
} ;
```

程序13-16演示了这个方法。Account类和CheckingAccount类与程序13-14中相同，增加的派生类InternationalAccount，没有引入额外的数据成员，只是引入了一个成员函数deposit()来满足新的客户要求。由于带有不同个数参数的deposit()消息的目标对象属于不同的类，因而不再有隐藏和重载问题。对象a1是带有一个参数的deposit()消息的目标对象，编译程序将调用基类的函数。对象a2是带有两个参数的deposit()消息的目标对象，编译程序将调用从CheckingAccount类派生而来的InternationalAccount类的函数。程序的运行结果如图13-13所示。

程序13-16 提高Account类的继承层次的例子

```
#include <iostream>
using namespace std;

class Account {                                    // base class
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
  double getBal()                                  // inherited without change
  { return balance; }
  void withdraw(double amount)                     // overwritten in derived class
  { if (balance > amount)
      balance -= amount; }
  void deposit(double amount)                      // inherited without change
  { balance += amount; }
} ;                                                // no changes to existing class

class CheckingAccount : public Account {           // derived class
protected:
  double fee;

public:
  CheckingAccount(double initBalance = 0)
  { balance = initBalance;  fee = 0.2; }

  void withdraw(double amount)                     // hides the base class method
  { if (balance > amount)
      balance = balance - amount - fee; }
} ;                                                // no changes to existing class

class InternationalAccount : public CheckingAccount {   // great!
public:
  InternationalAccount(double initBalance)
  { balance = initBalance; }

  void deposit(double amount, double fee)          // hides base method
  { balance = balance + amount - fee - CheckingAccount::fee; }
```

```
} ;                                                    // work is pushed to a new class

int main()
{
  CheckingAccount a1(1000);                            // derived class object
  a1.withdraw(100);                                    // derived class method
  a1.deposit(200);                                     // base class method
  InternationalAccount a2(1000);                       // new server object
  a2.deposit(200,5);                                   // derived class method
  cout << " Ending balances\n";
  cout << "   First checking account object:  "
       << a1.getBal() << endl;
  cout << "   Second checking account object: "
       << a2.getBal() << endl;
  return 0;
  }
```

```
Ending balances
  First checking account object:  1099.8
  Second checking account object: 1194.8
```

图13-13 程序13-16的输出结果

这是改进程序的一种非常有用的技术。不再破坏已有的类，也避免了使已有的客户代码无效的危险，而是从已有的类派生另一个类，让这个类只负责新的程序功能。使用C++继承方法是软件维护新方法的基础：它实现了通过编写新的代码而不是修改已有的代码来进行维护。

实际上，CheckingAccount类也需要进行一些修改。首先应将私有的数据成员fee改为受保护的，确保新的派生类InternationalAccount能够访问这个数据成员。另外一个办法是在CheckingAccount类中增加一个成员函数来获取这个数据成员的值，客户代码（这里是InternationalAccount类）将调用这个成员函数访问基类的数据。正如我们已经提到的，建议让几个数据成员可以被一两个派生类访问，而不要创建一系列的只供新的派生类使用的访问函数（这里只有一个派生类）。

避免修改已存在的CheckingAccount类的另一个途径是在设计程序时要有远见。为什么要将类的数据成员定义为private呢？根据面向对象程序设计的基本原则，有以下理由这样做：

- 不想让客户代码依赖于服务器类的数据名。
- 不想让客户代码直接对服务器类的数据进行操作从而让客户代码复杂化。
- 不想让客户代码知道不必知道的服务器设计。
- 让客户代码调用其名字清楚解释其操作的服务器方法。
- 让客户代码将处理低级的细节的任务推向服务器类。

注意，将服务器类的数据成员定义为protected而不是private，可以达到所有这些目标。正如我们在前面提到过，protected关键字与其他的访问权限修饰符private和public一样，与不同种类的类用户相关。对于通过继承链接的派生类，protected关键字如同public关键字一样，允许派生类直接访问基类的成员。对于没有通过继承链接的客户类，protected关键字如同private关键字一样，没有任何区别。如果程序可以通过继承来改

进，则应使用protected访问权限而不是private。

提示　要经常使用C++继承方法改进程序，将职责从客户代码推向新的派生类。将这种方法与创建许多较小类的不足进行权衡。

我们在这里讨论的是改进程序而不是从头开始设计程序。在程序设计中，位于类继承层次顶部的是某些重要的基类，它们包含许多派生类。由于有大量潜在的类用户，数据封装、信息隐藏、将任务推给服务器等问题都变得非常重要。对于这些重要的类，可能要使用private访问权限修饰符强迫派生类使用访问函数。在改进程序时，用于进一步产生派生类的类本身就位于继承层次的底部（CheckingAccount类是个好例子），这些类没有大量的依赖于它们的派生类，因而数据封装、信息隐藏、将任务推给服务器等问题，就随着依赖类数目的减少而变得不重要了。

第二个修改是CheckingAccount类的构造函数。我们增加了一个缺省的参数值以避免在客户代码中创建CheckingAccount类的对象时出现语法错误。这与我们在第12章中讨论的复合类的问题比较类似。在下一节，我们将讨论创建C++派生类对象的相关问题。

13.6　派生类的构造函数和析构函数

在创建一个派生类的对象时，它的基类部分和派生类部分都需要初始化。派生类对象的基类部分和派生类部分必须按严格的次序创建。理解这个次序对避免潜在的语法错误和性能问题很重要。

继承中的对象创建与类复合中的对象创建非常相似。对于类的复合，对象数据成员在执行复合类构造函数之前创建（它们的构造函数被调用）。如果没有合适的构造函数，试图创建一个复合对象就会产生语法错误。如果有合适的构造函数，复合对象的创建也可能会对程序性能造成一定的影响。

对于类的继承，派生类对象的基类部分总在对象的派生类部分创建之前及派生类的构造函数执行之前创建（它的构造函数被调用）。如果没有合适的基类构造函数，试图去创建一个派生类对象将会出现语法错误。如果有合适的基类构造函数，派生类对象的创建也可能会对程序性能造成一定的影响。

我们考虑程序13-6~程序13-9中的Point类，并对这个类进行改进，增加一个带有两个参数的通用构造函数：

```
class Point {                                   // base class
  int x, y;
public:
  Point(int xi, int yi)                         // general constructor
    { x = xi;  y = yi; }
  void set (int xi, int yi)
    { x = xi;  y = yi; }
  void get (int &xp, int &yp) const
    { xp = x;  yp = y; } } ;
```

这个改进的目的很明显：为客户代码在创建Point类的对象时提供更大的灵活性。现在，客户代码可以在创建对象时指定点坐标。这比创建未初始化的对象，稍后再通过调用set()成员函数进行初始化要好得多。

就程序13-6中的VisiblePoint类而言，基类中的这个改变不会要求它进行任何调整：

派生类没有受到影响。

```
class VisiblePoint : public Point {        // public inheritance
  int visible;
public:
   void show()
     { visible = 1; }
   void hide()
     { visible = 0; }
   void retrieve(int &xp, int &yp, int &vp) const
     { get(xp,yp);                         // base public method is accessible
       vp = visible; } } ;                 // derived private data is available
```

然而，受影响的是派生类VisiblePoint的客户代码，现在这个客户代码出现了语法错误。

```
int main ()
{ VisiblePoint b; int x, y, z;       // define a derived object: error
  b.set(20,40);  b.show();           // base, derived public functions
  b.retrieve(x,y,z);                 // call derived public function
  cout << " Point coordinates: x=" << x << "  y=" << y << endl;
  cout << " Point visibility:  visible=" << z << endl;
  return 0; }
```

对于对象而言，派生类对象的数据成员（这里是数据成员visible）在执行派生类的构造函数体之前被分配空间。但在派生类中描述的数据被分配空间之前，派生类对象的基类部分先被创建。这里的“创建”指的是基类数据成员（这里是Point类的x和y）被分配了空间，并调用了基类构造函数。

既然没有参数传递给基类的构造函数，就调用缺省的构造函数。由于基类Point提供了一个非缺省的构造函数，因而不能用系统提供的缺省构造函数。

这样，试图去创建一个派生类对象将会出现语法错误：调用的函数并不存在。注意程序13-6~程序13-9中很正常的客户代码现在出错了：

```
VisiblePoint b;              // no syntax error in previous versions
```

这个语法错误是由于给Point类增加了一个通用构造函数而引起的。在传统的程序设计语言中，增加新的代码可能会破坏已有代码的操作，但不会造成语法错误。而在C++语言中，这是一种新的程序不同部分之间的链接关系。

补救措施当然是为基类再提供一个缺省构造函数（或者系统提供的或者程序员定义的）。这个构造函数在派生类对象的基类部分被分配空间之后调用。

```
class Point {                        // base class
  int x, y;
public:
  Point()
    { x = 0;  y = 0; }               // now the client code is OK
  Point(int xi, int yi)              // general constructor
    { x = xi;  y = yi; }
  void set (int xi, int yi)
    { x = xi;  y = yi; }
  void get (int &xp, int &yp) const
    { xp = x;  yp = y; } } ;
```

现在，客户代码就不再有语法错误了，但Point类缺省构造函数所做的工作被浪费掉了，因为客户代码将VisiblePoint类的对象设置到了平面上所需的位置，或者显示或者隐藏。

```
VisiblePoint b;              // no syntax error
b.set(20,40);                // write over the base part of object
b.show();                    // set the derived part of object
```

创建一个派生类对象的相关事件序列如图13-14所示。首先，创建它的基类部分，接着调用基类缺省构造函数，再创建派生类部分，调用派生类的构造函数，最后执行客户代码中的下一条语句。

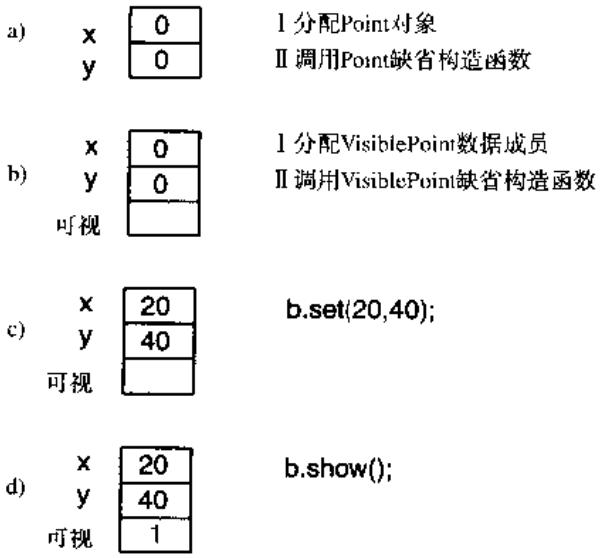


图13-14　分配并初始化一个派生类对象的步骤

这个例子的关键在于，基类部分被构造之后，基类缺省构造函数立刻初始化基类的数据域，只有在此之后才创建派生类部分及执行派生类的构造函数。如果客户代码需要将基类部分设置为某个特定的状态（而不是缺省状态），那么基类缺省构造函数的调用就被浪费掉了。

将任务从派生类的客户推到派生类构造函数可以改善这个设计。什么任务呢？即初始化包括其基类部分在内的派生类对象。在上面的程序段中，这个任务是由客户代码通过向派生类对象发送set()和show()消息来完成的。客户代码应该从这个任务中解脱出来。

派生类可以接受作为它的派生类的构造函数参数的数据，并利用这些数据初始化它自己的数据成员及其基类的数据。在派生类构造函数体内，这些参数可用来显式地设置基类部分的状态。

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view)   // parameters for data
   { set(xi,yi);  visible = view; }        // set base, derived fields
    . . . . } ;                            // the rest of the VisiblePoint class
```

现在，客户代码不需要显式地调用基类的成员函数set()和派生类的成员函数show()或hide()，而是在定义派生类对象时指定额外的参数。

```
VisiblePoint b(20,40,1);     // no need to call set() or show()
```

可以用不同的方式来看待VisiblePoint类构造函数中的set()函数调用。其中一种方式是编译程序首先试图在这个构造函数作用域之内寻找匹配的函数名，然后在VisiblePoint类作用域内寻找匹配，再接着在基类Point的作用域内寻找。另一种看待方式是set()函数属于基类。由于派生类对象“是一个”基类对象，set()函数也属于派生类。因此，set()函数的调用不需要目标对象，因为派生类对象（或其基类部分）是消息的目标。

第二种看待方式是为代码的阅读者着想，承认代码的编写者对编写得更快更感兴趣，而不是让代码更加易读。在编写代码时，代码的编写者知道set()函数所属的类；但代码的阅读者不得不在“一种看待调用的方式”和“另一种看待方式”之间进行选择。这意味着这段程序不是根据面向对象方法的基本原则编写的。根据这些原则，编写的客户代码（这里是VisiblePoint类的构造函数）应能让函数调用的名字显式地解释函数的行为。C++允许使用类作用域运算符来支持这种原则，从而将程序员编写代码时的想法传达给代码的阅读者。

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view)  // parameters for data
   { Point::set(xi,yi);                   // pass knowledge to maintainer
     visible = view; }
    . . . . } ;                          // the rest of the VisiblePoint class
```

明白这个意思吗？我们再次强调编程直觉问题。当然，传统的语言也提供了一些方法将设计人员的思想传达给代码的读者。但是，C++比传统语言要复杂得多。至少，编写C++代码的不同方法要比用传统语言编写代码的方法多。因此有更多的方法在代码中表达设计人员的思想，也更有必要在用C++编写的代码中表达设计人员的思想。一定要培养直觉，见机行事。

提示 要总是想办法将设计程序时的意图传达给客户端代码程序员及维护人员。C++允许在代码中而不是在注释中说明程序代码的意图。相当多的程序员将C++语言当做传统程序设计语言来使用，而不会利用它的这个优点来提高程序的质量。

13.6.1 在派生类构造函数中的初始化列表

在VisiblePoint类中增加一个构造函数，就将任务从VisiblePoint类的客户代码推到VisiblePoint代码中。但这并未解决调用基类构造函数所造成的浪费。

无论如何，都要为派生类的基类部分调用基类缺省构造函数，这个调用在对象的基类部分被分配空间之后立即执行。由于在执行构造函数体时，基类部分的域在派生类构造函数中被重置，因而对基类缺省构造函数的调用就没有用。

如果基类有非缺省的构造函数，派生类构造函数可以调用这个非缺省的基类构造函数，而不调用缺省的基类构造函数。这样可以避免浪费函数调用。

注意，总是在为基类部分分配空间之后和在调用派生类构造函数之前调用基类构造函数。问题只是调用哪个构造函数——缺省的构造函数还是非缺省的构造函数。

为了调用带参数的非缺省的基类构造函数，C++支持初始化列表语法，这与我们在类复合中用来协调构造函数的调用的成员初始化列表的语法类似。

```
class VisiblePoint : public Point {
```

```
  int visible;
public:
   VisiblePoint(int xi, int yi, int view) : Point(xi,yi)     // list
    { visible = view; }                  // no call to set()
    . . . . } ;                          // the rest of class VisiblePoint
```

这两种形式的初始化列表之间的区别非常重要。在类复合中，初始化列表包括类数据成员名形式的对象元素名。在继承中，初始化列表包含派生类对象元素名，其形式是基类名而不是某个数据成员的名字。

这两种形式的初始化列表之间的相似处在于元素构造函数调用的时间。在类复合中，在为元素数据成员分配了空间之后立刻调用。在类继承中，在为派生类对象的基类部分分配了空间之后立即调用。它们都是在执行类构造函数体（容器类或派生类）之前调用的。如果派生类对象的基类部分还包含其他类的元素，或者如果复合对象的元素含有基类时，这个过程将递归地进行。

总的来说，在C++中创建一个派生类对象时，首先创建其基类部分，然后调用基类构造函数，再创建派生类部分，执行派生类构造函数体。

构造函数调用中冒号后的参数被传递给了基类的构造函数。这些参数既可以是从客户代码传递给派生类的构造函数的参数（如上一个例子），也可以是字面值，甚至还可以是函数调用。对此并没有任何限制。

如果基类成员需要一个缺省的（没有参数的）构造函数来初始化派生类对象，那么这个缺省的构造函数既可显式地调用，也可隐式地调用。例如，VisiblePoint类的对象要将其基类部分初始化为屏幕上的原点坐标，那么VisiblePoint类的构造函数可以写成如下形式：

```
class VisiblePoint : public Point {
  int visible;
public:
 VisiblePoint(int view) : Point()    // call to default constructor
 { visible = view; }                 // no call to set()
   . . . . } ;                       // the rest of class VisiblePoint
```

另一方面，没有必要显式地调用基类的构造函数。即使没有初始化列表，编译程序也会自动地激活缺省的基类构造函数。

```
class VisiblePoint : public Point {
  int visible;
public:
 VisiblePoint(int view)      // implicit call to default constructor
  { visible = view; }        // no call to set()
   . . . . } ;               // the rest of class VisiblePoint
```

在创建派生类对象时，这两个版本的派生类构造函数会产生相同的事件序列：为对象的基类部分分配空间，调用缺省的基类构造函数，为对象的派生类部分分配空间，调用派生类的转换构造函数。

可以混合使用两种列表，这样也可以使用成员初始化列表语法来初始化派生类的数据成员。

```
class VisiblePoint : public Point {
  int visible;
public:
```

```
VisiblePoint(int xi, int yi, int view)
 : visible(view), Point(xi,yi)          // what is called first?
 { }                                    // a popular C++ idiom
. . . . } ;                             // the rest of class VisiblePoint
```

数据成员总是按照它们在类定义中的次序来创建的。对于一个派生类，基类部分的定义隐式地成为类定义中的最开始部分——出现在派生类成员的定义之前。尽管初始化列表为以上形式，但仍然首先调用基类构造函数，只有在调用后才初始化派生类数据成员，派生类的构造函数体（如果有的话）总是在最后执行。

在设计一个派生类时，很少不使用构造函数和初始化列表。

在这个例子中，派生类的构造函数体是空的。在构造函数初始化列表中初始化所有的派生类数据成员并没有什么优越之处，但仍被人们普遍使用。出于某种难以解释的原因，如果派生类的构造函数为空会让许多C++程序员觉得这是个好的设计。

总而言之，如果是以下情况，则没有必要在派生类的构造函数设计中使用初始化列表：

1) 基类没有构造函数（则当创建派生类对象的基类部分时，调用系统提供的缺省构造函数）。

2) 基类有一个程序员定义的缺省构造函数（在创建派生类对象的基类部分时被调用），而且派生类的构造函数（如果有的话）不会改变这个缺省构造函数所创建的派生类对象的基类部分的状态。

如果基类有一个非缺省的构造函数，要注意区分以下两种情况：

1) 基类没有缺省的构造函数：那么派生类构造函数必须使用初始化列表语法来激活非缺省的基类构造函数，以避免在定义派生类对象时出现语法错误。

2) 基类还有一个程序员定义的缺省构造函数：那么派生类构造函数并不一定要使用初始化列表。首先调用缺省的基类构造函数，然后派生类构造函数在函数体内覆盖了缺省构造函数的行为。使用初始化列表语法调用合适的基类非缺省的构造函数会比较好。

派生类可以没有程序员定义的构造函数吗？当然可以。这意味着派生类的基类部分和派生类部分都不需要初始化。但是，如果这样的事情真的发生了，请再次检查设计，因为可能遗漏了什么东西。

13.6.2 继承中的析构函数

下面的例子虽然不是很好，但它能说明使用派生类的析构函数的相关问题。

我们要设计一个Address类存放每个人的姓名和e-mail地址。由于继承是一种对程序中的类进行组织的强有力机制，我们想从另一个更简单的类Name派生Address类，这个Name类包含了e-mail地址拥有者的姓名。基类Name有一个数据成员data，这个数据成员指向动态分配的字符数组。类构造函数动态地为对象分配内存，并将参数字符串拷贝到堆内存中。在撤销对象之前，析构函数把字符串内存返回到堆中。get()成员函数返回指向姓名的指针。

程序13-17实现了这个设计，由于这个例子是用来说明基类和派生类的动态内存管理问题的，我们尽量让它简单些。因此我们没有为这些类实现拷贝构造函数和赋值运算符。潜在用户也不必创建Name对象：因为Name类只用作Address类的基类。为了防止潜在的灾难性问题，我们将Name类构造函数定义为Protected，以防止用户创建Name对象。我们不能将它定义为private，否则Address类的对象将不能初始化其基类部分。但这没有问题，因为对

于潜在用户来说，protected如同private一样。对于Address类，我们将拷贝构造函数和赋值运算符定义为private，以防止潜在的问题。

程序13-17　为负责动态内存管理的类使用继承

```
#include <iostream>
using namespace std;

class Name {                                          // Base class
  char *name;                                         // dynamic memory management
protected:
  Name(char nm[]);                                    // prevent using the objects
public:
 ~Name();                                             // return dynamic memory
  const char* get() const; } ;                        // access the contents

Name::Name(char nm[])
{ name = new char[strlen(nm)+1];                      // allocate heap space
   if (name == NULL) { cout << "Out of memory\n";  exit(1); }
   strcpy(name,nm); }                                 // initialize heap memory

const char* Name::get () const
{ return name; }                                      // access private data

Name::~Name()
   { delete [] name; }                                // return object data

class Address : public Name {                         // Derived class
  char *email;
  Address(const Address&);                            // no value semantics
  void operator = (const Address&);
public:
  Address(char name[], char address[]);               // allocate heap space
 ~Address();
  void show() const; } ;                              // display object data

Address::Address(char nm[], char addr[]) : Name(nm)
{ email = new char[strlen(addr)+1];
  if (email == NULL) { cout << "Out of memory\n";  exit(1); }
  strcpy(email,addr); }

Address::~Address()                                   // return object memory
{ delete [] email; }

void Address::show() const                            // display object data
{ cout << " Name:   " << Name::get() << endl;
  cout << " Email: " << email << endl << endl; }

int main ()
{
  Address x("Shtern", "shtern@bu.edu");               // client code
  x.show();
  return 0;
  }
```

程序的输出结果如图13-15所示。

Name类的构造函数（基类的构造函数）为基类分配内存，并拷贝数据；Address类的构造函数（派生类的构造函数）为派生类分配空间，并拷贝数据。

Address类构造函数在执行其函数体之前，将数据值传递给Name类的构造函数。实例化

一个Address类的对象需要按照以下步骤：

```
Name:  Shtern
Email: shtern@bu.edu
```

图13-15 程序13-17的输出结果

1) 为派生类对象的基类部分分配内存（指针name）。

2) 调用基类构造函数，分配并初始化name指针指向的堆内存。

3) 为派生类对象的派生类部分分配内存（指针email）。

4) 调用派生类的构造函数，分配并初始化email指针指向的堆内存。

析构函数的调用次序与构造函数的调用次序相反，当撤销一个派生类对象时，首先执行派生类的析构函数，之后撤销派生类部分的数据成员。然后调用基类的析构函数，随后再撤销对象的基类部分。下面是发生的行为列表：

1) 调用派生类的析构函数，将email指针指向的堆内存还回给系统。

2) 撤销对象的派生类部分，并将其内存（email指针）还回给系统。

3) 调用基类的析构函数，将name指针指向的堆内存还回给系统。

4) 撤销对象的基类部分，并将其内存（name指针）还回给系统。

由于一个类的析构函数可以没有参数，因此程序员不必协调析构函数的调用，只要保证有析构函数即可。如果没有实现任何析构函数将会导致内存泄漏。

对象的基类部分不应先撤销，因为它是对象派生类部分的服务器。基类数据成员对于保持派生类部分的数据成员完整性可能是必要的。

也可以在Address构造函数和析构函数中加入很多允斥对两个类的动态内存管理。但是，只管理一个类的内存是好的模块化处理。

由于这是一个小例子，类之间的关系并不重要。但从Name类派生Address类扩展了关系的概念。一个地址并不是一个名字，但是使用继承就暗示了这种关系。更合适的说法是一个地址有一个名字，这提示我们使用复合关系。在下一章，我们会更加详细地讨论继承和复合之间的权衡。

13.7 小结

本章，我们继续讨论了C++类之间的关系。继承允许将一个类作为另一个类的基类使用。通过使用继承，派生类将继承基类的所有数据成员和成员函数。通常，派生类还要增加一些其他的数据成员和成员函数。有时，派生类将重定义从基类继承而来的成员。

继承为模块化设计提供了一种有效途径。设计服务器类不是一蹴而就，可以先创建并调试好一个基类，然后以派生类的形式增加其他的功能。

使用继承也有助于程序的改进。不用改变已有的代码，而是增加新的代码到客户代码中，然后通过创建新的派生类来支持新的客户代码的需要，而不用修改已有的服务器类。

与类复合相似，使用继承要求掌握大量的新的语法细节。C++有很多种方式实现继承，经常可以让我们用不止一种方法实现设计。但是，这也意味着不谨慎地使用继承可能会使程序更加复杂。

使用继承作为设计工具，我们可以扩展练习软件工程现代模式的机会，即将职责从客户代码推向服务器类，并将设计者的意图通过程序代码传达给程序员及维护人员。这是编写代码的新方法，需要转换一下直觉。一定要坚持培养我们用C++进行程序设计的直觉，它是我们程序设计技能的一个重要部分。

本章所讨论的只是C++继承的一部分，下一章我们将介绍其他使用继承的方法。

第14章 在继承和复合之间进行选择

本章，我们将讨论许多有关继承和复合的例子。首先用一个例子来比较继承与其他程序设计方法之间的区别。

我们通过用不同的设计方案实现同一程序来对这些设计方案作一些比较。在这里，“设计”一词的意思与它在本书其他地方的意思相同，指的是决定程序由哪些部分（类）组成及这些部分如何进行任务划分（数据成员和成员函数）等问题。在比较这些不同设计方案时，我们用在第1章中所讨论的通用准则对它们各自的效果进行评价，准则包括将任务从客户类推给服务器类、以调用服务器方法的形式编写自说明的（self-documented）客户代码、去掉类之间的链接等。我们同时还使用其他与低层相关的标准：如封装、信息隐藏、耦合度与内聚性等。

所有这些方法的目的都是为了提高程序的可读性。在本章，评价程序设计质量的标准之一是易编程性。这与我们在第1章所提出的基本原则有所不同。在第1章我们强调了可读性，并认为易编程性是通过可读性获得的，因此应该不加考虑。毕竟，只用编写程序一次，而且真正的输入代码的时间只占阅读程序所用时间的极小部分，而我们在对程序进行调试、测试、集成、重用或修改时要花大量的时间阅读程序。

在使用继承时将重点从可读性转移到易编程性是不可避免的，因为继承是一种以提高易编程性为目的的技术。服务器类的设计者从基类派生出服务器类，其目的是方便地实现服务器类，而不是更好地为客户代码服务。理想情况下，客户代码的设计者不用关心服务器类是全新设计的，亦或是从某个基类派生而来，只要服务器类能满足客户代码的需要即可。

这只是理想状态而已，实际情况和C++编程的美好愿望是不同的。对于客户代码和维护人员而言，使用（以提高服务器代码编写性为目的的）继承将使程序的可读性降低。在下一章，我们将介绍如何使用继承让客户代码更为简单。

为了讨论类之间的链接关系，我们将使用统一建模语言（UML）来描述应用程序中类之间的关系。今天，使用UML已被认为是面向对象设计和实现的关键。许多组织在他们的面向对象工程中都接受使用UML。大量的事实说明了UML很丰富，但是尚未有确凿的证据说明使用UML可以使面向对象项目成功。UML是策略上和技术上的折中产物，而不是突破性发展的结果。设计它的目的是为了统一几个早期的不同的面向对象设计符号（notation），并添加一些可以用来更加详细地描述对象关系的功能。但是，各种组织的每个成员都试图在UML中添加他们喜欢的符号系统。因此，这个语言充斥了过多的功能，很难学习。

这很可惜，因为语言不应该具有强迫性。UML应允许开发人员比较容易地相互交流思想和相互理解。对于初学者，他们编写的程序代码往往模棱两可，使人迷惑不解，对此，应有一个编译程序提醒设计者。但UML中没有提供这些服务。用UML所生成的图过于复杂。我们使用UML（及它的以前版本）的经验表明，在掌握一门面向对象语言之前学习UML会浪费很多时间。而且，该建模语言非常庞大，并提供了很多可选的设计方案。因此在学习面向对象语言的同时最好不要尝试着去学习UML，它不会加快学习的进程。但UML（或者以前的类似工具）的基本概念可以用来描述C++实现的面向对象设计。

因此，在本章，我们将介绍UML的基本表示方法，将它作为描述工具比较继承和复合的使用。同时也将使用UML符号来表示程序中对象之间的关系。我们在本章讨论的例子都非常大，适合于不同的设计和实现方法，使用UML将有助于理解程序设计中高层次的问题。

14.1 选择代码重用的方法

本节我们将讨论使用继承和复合的相对优缺点。继承和复合都是客户-服务器关系。派生类是基类的客户，基类是派生类的服务器。复合类是其组件类的客户，组件类是复合类的服务器。这样，我们将可以分析出不同设计所实现的C++程序之间的相似处。

不论是使用复合还是其他设计技术，不同设计方法的一个共同特点是客户类和服务器类之间的任务划分。服务器类应在设计客户类之前实现。因此，本节所讨论的各种方法不仅适用于程序开发，也同样适用于程序改进。

14.1.1 类之间的客户-服务器关系的例子

作为客户-服务器关系的一个简单例子，我们使用了一个Circle类，它有一个表示圆半径的数据成员。这样，客户代码可以通过发送消息访问Circle对象的内部数据。

```
Circle c1(2.5), c2(5.0);                    // set the value of radius
double len = c1.getLength();                // compute circumference
double area = c2.getArea();                 // access internal data
c1.set(3.0);
double diam = 2 * c1.getRadius();
```

为了支持以上的客户代码，Circle类至少要有5个公共成员函数：

- 带有一个整型参数的构造函数。
- 返回圆周长的getLength()方法。
- 返回圆半径的getRadius()方法。
- 改变圆半径的set()方法。
- 返回圆面积的getArea()方法。

值得注意的是，特定的客户代码决定了服务器类的形式，这不是C++程序设计的惟一模式。如果强调软件组件的重用性，服务器类常被设计成类库，以尽量满足最大数目客户的需要。为了达到这个目的，服务器类提供其设计者认为可以满足最大数目客户需要的服务。其结果是，某些客户在使用这些通用类时不得不做更多的工作。

本书中我们不想把重点放在库类的设计上。设计这些类时，要提供对内部数据表示的访问途径，并让客户端代码程序员能方便地操纵这些数据。

C++程序设计的第二种模式，即支持客户-服务器关系的模式，更具有挑战性。它要求服务器程序识别客户需求，并提供方法满足这些需求，而不是将信息留给客户代码处理。

注意，我们将消息发送给服务器对象去访问内部数据表示。无论类方法具有什么功能（如将圆半径乘以2再乘以PI以计算圆的周长），它将代表没有访问权限的客户代码访问内部数据（例如，半径）。在本例中为了满足客户的这种需要，Circle类应为如下形式：

```
class Circle                                // original code for reuse
{ protected:                                // inheritance is one of the options
    double radius;                          // internal data
  public:
```

```
  Circle (double r)                // support for initialization
  { radius = r; }
double getLength () const          // compute circumference
{ return 2 * PI * radius; }
double getArea () const            // compute area
{ return PI * radius * radius; }
double getRadius() const
{ return radius; }
void set(double r)
{ radius = r; } };                 // change size
```

如果想避免因写浮点数而出现的错误（或者如果希望更多地练习使用初始化列表），可以使用如下形式的Circle类。

```
class Circle                                    // original code for reuse
{ protected:                                    // inheritance is one of the options
    const double PI;                            // it must be initialized in the list
    double radius;                              // internal data
  public:
    Circle (double r) : PI (3.1415926536)       // initializer list
    { radius = r; }
    double getLength ()                         // compute circumference
    { return 2 * PI * radius; }
    double getArea ()                           // compute area
    { return PI * radius * radius; }
    double getRadius() const
    { return radius; }
    void set(double r)
    { radius = r; } };                          // change size
```

注意，我们在使用常量而不是数值文字时，只应用了一个原则：在代码的不同地方输入同一数字时可能会有输入错误；而没有应用另一个原则：在维护的时候方便修改。因为，除非是有出人意料的科学突破，否则PI的值一般情况下是不会改变的。另外也要注意，每次调用getLength()方法时，我们将PI乘以2。这些不是严重的缺点，它们表明，在本例中将PI定义为一个常量的真正目的是：再次说明初始化列表中不仅可以包括构造函数的参数，还能包括字符参数。

最后，注意PI定义为Circle类的局部数据成员。如果应用程序中的其他类也需要这个值，它们或者自己再定义或者从Circle类中获取。为了方便处理，可将该常量数据成员定义为public。

```
class Circle                                    // original code for reuse
{ protected:                                    // inheritance is one of the options
    double radius;                              // internal data
  public:
    const double PI;                            // it must be initialized in the list
  public:
    Circle (double r) : PI (3.1415926536)       // initializer list
    { radius = r; }
  . . . } ;                                     // the rest of class Circle
```

在这里使用了一个流行的技巧，即将公共数据成员在各自的公共段中进行定义，这样比较引人注目。如果我们将PI与类的成员函数在一起定义，也许PI会丢失。

现在Circle类已经定义好了——但是考虑一下，真的已经好了吗？在Circle类的设计

中，每个Circle对象分别为PI分配内存。同时，每个对象中PI的值是相同的。为每个Circle对象分配这样的内存是一种浪费。使用非面向对象语言的程序员不需要考虑这些问题，但是C++程序员总是要处理这些问题。除非已经建立了良好的直觉，否则我们建议一定要谨慎地检查每个数据成员的使用模式。当类的某个数据成员在每个对象中的值都相同时，可使用静态数据成员。下面的Circle类为所有对象中的PI只分配了一段内存。

```
class Circle                          // original code for reuse
{ protected:                          // inheritance is one of the options
    double radius;                    // internal data
  public:
    static const double PI;           // it must be initialized
  public:
    Circle (double r)                 // initializer list
    { radius = r; }
    double getLength () const         // compute circumference
    { return 2 * PI * radius; }
    double getArea () const           // compute area
    { return PI * radius * radius; }
    double getRadius() const
    { return radius; }
    void set(double r)
    { radius = r; } };                // change size

  const double Circle::PI = 3.1415926536;
```

正如我们看到的，初始化一个静态数据成员不需要使用成员初始化列表。静态数据成员可以在定义时初始化，这可以在与类成员函数所在的相同的文件中实现。与成员函数相似，类作用域运算符指定该数据成员所属的类。如果想在另一个类中再定义数据成员PI，它们之间不会产生名字冲突，因为它们属于不同的类。

这个例子再次表明，C++程序员在进行程序设计时应考虑不同方面，而不要局限于某一部分，否则会做出导致冗余甚至错误的结论。

一定要注意在编写C++代码时要多方面地考虑问题，思路不要太窄。

现在类Circle已经设计好了。下面我们考虑Cylinder类的客户代码的需求，Cylinder类的数据成员分别表示圆柱体的半径和高。

```
Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);        // initialize data
double length =cyl1.getLength();              // similar to Circle
cyl1.set(3.0);
double diam = 2 * cyl1.getRadius();           // no call to getArea()
double vol = cyl2.Volume();                   // not in Circle
```

即使Circle类和Cylinder类不同，但它们有相似的内部结构（数据成员radius），提供了相同的名字和相同接口的服务。例如，getLength()。因此，可以在Cylinder类设计中重用Circle类。

虽然我们努力使例子尽量小，以集中注意力在设计问题而不是设计细节上，但是本例也可以反应出，Circle中已有的某些服务在Cylinder类中应该是无效的，例如getArea()方法。另一方面，Cylinder客户需要的某些服务对于Circle的客户而言也是无效的，如Volume()方法。这对于大多数重用上下文都很典型——重用了其中某些已有的服务，抑制或者忽略了一些服务，而且又可能添加了若干新的服务。

下面我们假设已有了Circle类的代码，而Cylinder类还没有设计。这两个类之间的相似性暗示我们可以使用Circle类的代码创建Cylinder类，这样就最大程度地重用了可用代码。

对许多人来说，这种相似性正是使用继承的决定因素。这也太简单了。继承在工业中已经使用得过多了。在重用时继承不应该作为第一选择，至少它应该在与别的代码重用方法作比较之后再被选中。那么如何选择最合适的方法呢？

重用代码的数量和方便性应是首要的标准。其次的两个标准是要编写的新代码数量与测试情况数量。这个例子很小，因此区别不是很明显，但可以用来说明选择时要注意的问题。

一般来说有四种代码重用方法：思路重用（如从头开始编写代码）；编写一个新的类，这个类的方法使用（借用）已存在的类（服务器）；编写一个新的类，这个类从已存在的类继承而来，其对象为客户提供基本服务；使用继承并重新定义某些方法。在这个例子中，这些方法的使用情况如下：

1) 思路重用：为Cylinder从头开始编写新的代码，使用编辑器把Circle代码中的radius、getLength()及其他成员函数拷贝给Cylinder类，并增加新的Cylinder代码来处理Circle类没有提供的服务。

2) 借用服务：假设每个Cylinder对象中包含“有一个”Circle对象，将Cylinder类作为一个复合类来设计。Circle类型的对象作为Cylinder类的一个数据成员，Cylinder方法（如getLength()）使用相同的成员名发送消息给Circle组件。

3) 将已存在的类作为基类来继承：假设每个圆柱对象“是一个”圆（只不过添加了一些属性，可能还删除了一些属性），可将Cylinder 类设计为Circle类的派生类。对于继承的方法不需要再编写代码，例如getLength()。因为每个Cylinder对象能够响应从其基类Circle中继承的消息。

4) 继承但重新定义某些方法：该方法允许以一种新的方式去处理已有的操作。例如，计算圆柱面积的方法与计算圆面积的方法不同。

下面，我们分别使用上述的各种方法来实现Cylinder类，并对每种方法的相对优缺点进行分析。

14.1.2 运用智力的重用：重做

思路重用是非面向对象程序设计中常用的方法。而在面向对象程序设计中，程序员们似乎都更加愿意使用继承与复合，而不屑于使用这么低级的代码重用方法。

使用该方法依赖于过去。如果有类似任务的开发经验，就将以前所写的代码重新写一遍并进行适当的修改，使之满足新的需要。在本例中，假设我们最近编写并测试了Circle类，而现在的任务是编写Cylinder类。我们命名这种方法为思路重用是因为，重用的是我们在相似代码的工作中积累的知识。

程序14-1是Cylinder类的设计，它使用了Circle类的设计。我们再次定义了Circle类的数据部分（即radius数据成员），并增加了Cylinder类所要求的数据（height数据成员）。我们也再次定义了构造函数，并增加了初始化height数据成员的参数和代码。还将可重用的方法一字不漏地拷贝过去（getLength()及其他方法），并增加Circle类所没有的Cylinder方法（增加了Volume()方法）。我们没有关心Cylinder类不需要的Circle方法（getArea()方法）。程序的执行结果如图14-1所示。

程序14-1　思路重用代码例子

```
#include <iostream>
using namespace std;

class Cylinder                                          // new class Cylinder
{ protected:
    static const double PI;                             // from class Circle
    double radius;                                      // from class Circle
    double height;                                      // new code

  public:
    Cylinder (double r, double h)                       // from Circle plus new code
    { radius = r;
      height = h; }                                     // new code

    double getLength () const
    { return 2 * PI * radius; }                         // from class Circle

    double getRadius() const                            // from class Circle
    { return radius; }

    void set(double r)                                  // from class Circle
    { radius = r; }

    double getVolume() const                            // no getArea()
    { return PI * radius * radius * height; }           // new code
   } ;

const double Cylinder::PI = 3.1415926536;

int main()
{
 Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);                 // initialize data
 double length = cyl1.getLength();                      // similar to Circle
 cyl1.set(3.0);
 double diam = 2 * cyl1.getRadius();                    // no call to getArea()
 double vol = cyl2.getVolume();                         // not in Circle
 cout << " Circumference of first cylinder: " << length << endl;
 cout << " Volume of the second cylinder:   " << vol << endl;
 cout << " Diameter of the first cylinder:  " << diam << endl;
 return 0;
  }
```

```
Circumference of first cylinder: 15.708
Volume of the second cylinder:   589.049
Diameter of the first cylinder:  6
```

图14-1　程序14-1的输出结果

现有的Circle类的大部分代码（数据成员和方法）一字不漏地复制给了Cylinder类，不需要的方法则删除。在Cylinder类中还增加了Circle类中没有的新的数据成员和方法。这个程序还要进行测试。如果是使用文本编辑器复制而不是输入现有代码，则Circle代码的测试应该很少。由于Circle的函数接口没有改变，已有的Circle类的测试结果也可被

Cylinder类重用。

这种代码重用方法效率很高。包括主管在内的所有人都会为我们的代码开发速度大吃一惊。如果他们知道我们是依赖于以前的经验，就不会觉得那么突兀。从另一方面来说，正是得益于我们在相似系统中的开发经验，才会有这样的效率，因此这种经验是开发小组最宝贵的财富。

但从软件工程的观点来看，这种方法有一个很严重的不足。不足之处是什么呢？Circle类和Cylinder类彼此是相互关联的，它们有相同的数据成员和成员函数，而只有Cylinder类的设计者才知道这种关联，维护人员则可能会忽略这种关联，这样就会导致维护时的错误。

14.1.3 借助服务重用

编写C++程序时一个好的做法是，某些对象把消息发送给实际生活中也相关的其他对象。把消息发送给其他对象常被称为那个对象借用服务（buying the service）。

注意，我们这里说的是在程序中某些对象发送消息给另外一些对象，而不是对象之间互相发送消息。从语法上讲，在同一程序中类A的对象把消息发送给类B的对象，类B的对象也把消息发送给类A的对象，这是可能的。C++允许这种“双工”式的合作。而且，这种体系结构在实时程序中可能非常有用。但大多数情况下，它会使导致其设计出现不必要的复杂：这些协作类之间任务划分变得复杂，类之间的链接关系也更为复杂。因此在大多数类互相协作的情况下，某个类充当客户类，而另一个类充当服务器类。当客户类的方法把消息发送给服务器类的对象时，我们说这个类“借用”了另一个类的服务。

客户方法访问服务器对象并把消息发送给服务器对象时，有三种情况：

- 在客户方法中将服务器对象定义为局部变量。
- 在客户类中将服务器对象定义为数据成员。
- 将服务器对象作为客户方法的参数。

从类通信的角度看，第一种情况是最有利的：只有一个客户函数（服务器对象在此函数中定义）可以访问这个服务器对象。只要有可能，我们都应该选择这种类型的客户-服务器关系。但这又常常不太可能，因为通常情况下服务器对象还需要被其他的客户类方法或其他类访问。

第二种情况次优：客户类的所有成员函数都可以访问服务器对象。如果可以选择，宁愿使用这种类型的客户-服务器关系，也不愿将服务器对象作为客户类方法的参数。

从协作类之间的通信角度而言，第三种情况最为复杂：参数对象一方面是接受它作为参数传递的函数的服务对象，另一方面又是调用该服务器函数的服务对象。通常情况下，我们避免使用这种方法，而采用第一种方法（只被客户的一个方法访问）或第二种方法（只被一个客户类的方法访问）。

从代码重用的角度而言，第三种客户-服务器关系允许创建这样的服务器类：它通过提供已有类的服务满足客户的需要。

对于Circle类和Cylinder类之间关系的例子，要在这两个类之间建立客户-服务器关系，使得Circle类对象是Cylinder类的一个成员。在设计时我们尽量将数据成员定义为private成员（或protected成员），使Cylinder类的客户不能直接使用Circle类的服务。要想将这些服务提供给客户（如getLength()），Cylinder类得请求它的Circle类的数据成员去处理。

该设计如程序14-2所示，其输出结果与程序14-1相同。给出了Circle类的显式定义，Cylinder类定义了一个Circle类的数据成员和其他数据成员（本例中指的是height数据

成员）。如果这个Circle类的数据成员是public，那么Cylinder类的客户代码将访问它。

```
class Cylinder                                  // new class Cylinder
{ protected:
    double height;                              // new code
  public:
    Circle c;                                   // no PI, no radius
  public:
    Cylinder (double r, double h)               // from Circle plus new code
        : c(r)                                  // initializer list (no PI)
    { height = h; }                             // new code
    double getVolume() const                    // no getArea()
    { double radius = c.getRadius();            // new code
      return Circle::PI * radius * radius * height; }
  } ;                                           // no getLength(), getRadius(), set()
```

这个Cylinder类的成员函数很少。它没有必要为客户代码实现getLength()、getRadius()、set()等方法，因为客户代码可以将这些消息发送给Cylinder类的公共数据成员c。

```
Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);      // initialize data
double length = cyl1.c.getLength();         // use Circle data member
cyl1.c.set(3.0);
double diam = 2 * cyl1.c.getRadius();       // no call to getArea()
double vol = cyl2.getVolume();              // not in Circle
```

该设计的不足是它的数据是public。客户使用Cylinder类的数据成员名c，并建立了与Cylinder类之间的依赖。影响设计质量的因素是Cylinder类与客户代码之间的任务划分。这里，Cylinder类非常轻松，它所做的事是提供getVolume()方法，而不用考虑其他任务。谁知道getLength()、getRadius()、set()方法属于哪个类呢？客户代码知道这些方法，而服务器类Cylinder不知道。我们又是如何知道这种任务划分方式的呢？因为可以看到，是客户代码而不是Cylinder类发送了这些消息。

请注意，我们不是抱怨客户代码中糟糕的链式函数调用语法。它的确很糟糕，但是我们抱怨的是客户代码设计人员拥有过多扩展的责任。设计人员（和代码的维护人员）需要了解两个类的服务，即Circle类和Cylinder类，而不是仅仅了解一个类Cylinder的服务就可以了。在这个例子中，可以将这两个类的定义方便地放在一起。但是在实际生活中，它们是可以分开的，也可能不仅是两个彼此相关的类。在现实生活中，没有证据表明这些类（即Circle和Cylinder）是彼此相关的。

因此，我们认为程序14-2是一个借用服务的更好例子。在Cylinder 类中，Circle类数据成员不是public成员（它是protected成员，对于客户代码而言，它与private成员一样不能直接访问）。这样，是Cylinder类而不是其客户知道getLength()、getRadius()、set()方法属于哪个类。Cylinder类定义了一组单线函数（one-liner）——这些成员函数惟一任务是转过来把同名消息发送给Cylinder的数据成员c。

程序14-2　通过借用数据成员（类复合）服务重用代码的例子

```
#include <iostream>
using namespace std;

class Circle                                    // original code for reuse
```

```
{ protected:                                    // inheritance is one of the options
    double radius;                              // internal data
  public:
    static const double PI;                     // it must be initialized

  public:
    Circle (dcuble r)                           // conversion constructor
    { radius = r; }

    double getLength () const                   // compute circumference
    { return 2 * PI * radius; }

    double getArea () const                     // compute area
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                          // change size
  const double Circle::PI = 3.1415926536;

class Cylinder                                  // new class Cylinder
{ protected:
    Circle c;                                   // no PI, no radius
    double height;                              // new code

  public:
    Cylinder (double r, double h)               // from Circle plus new code
        : c(r)                                  // initializer list (no PI)
    { height = h; }                             // new code

    double getLength () const
    { return c.getLength(); }                   // from class Circle

    double getRadius() const                    // from class Circle
    { return c.getRadius(); }

    void set(double r)                          // from class Circle
    { c.set(r); }

    double getVolume() const                    // no getArea()
    { double radius = c.getRadius();            // new code
      return Circle::PI * radius * radius * height; }
   } ;

int main()
{
 Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);         // initialize data
 double length = cyl1.getLength();              // similar to Circle
 cyl1.set(3.0);
 double diam = 2 * cyl1.getRadius();            // no call to getArea()
 double vol = cyl2.getVolume();                 // not in Circle
 cout << " Circumference of first cylinder: " << length << endl;
 cout << " Volume of the second cylinder:   " << vol << endl;
 cout << " Diameter of the first cylinder:  " << diam << endl;
 return 0;
  }
```

这个例子既支持了数据封装也支持了任务划分。客户代码只需要知道其服务器Cylinder类，而不需要了解Circle类（一个服务器类的服务器）的设计。在维护时，这些类的关系十分清晰。

这种代码重用方法甚至比从头开始重写程序方法还要快。需要的测试也更少一些，测试单线函数十分简单，不涉及基类及派生类之间的继承和耦合度问题。

当Cylinder类需要访问Circle数据成员时可能会出现问题。因此，Circle类提供Cylinder类所需的访问方法十分重要。有些程序员不喜欢这些单线函数，它们虽然简单但比较烦人。下一节中的继承可以消除这个问题。

14.1.4　通过继承的代码重用

通过继承重用代码是目前最为流行的方法。通常，大部分的基本服务可以“原封不动”地被继承，只需要少量的添加或修改。这样得到的方法可以很好地工作，而且可以消除类复合中常出现的单线方法。

程序14-3中的例子，通过将Circle类作为派生类Cylinder的基类来重用代码。由于其客户代码与程序14-1和程序14-2相同，因此理所当然程序的输出结果也与图14-1相同。

在程序14-2中，我们假设圆柱“有一个”圆，因此Cylinder类实现了两个类通用的方法，以把消息传送给数据成员Circle类。而在这里，我们假设圆柱“是一个”圆。

派生类Cylinder的客户能比较容易地访问其基类的服务。客户代码在调用这些服务（如getLength()）时就好像这些服务是在Cylinder类中定义的一样。Cylinder类的设计也很简单，只需要定义其基类Circle中所没有的功能。这与将Circle类作为public数据成员的复合的使用一样容易。与程序14-2中将Circle类作为非公共数据成员的复合比起来，也肯定要简单得多。对于类复合，Cylinder类必须为每个Circle方法实现一个单线方法，以供Cylinder类的客户代码访问。而对于继承，则不需要这些单线方法。

派生类的构造函数所使用的初始化列表与类复合的初始化列表相似——用程序14-3中基类名代替程序14-2中的数据成员名。还记得初始化列表的作用吗？既然Circle类没有缺省的构造函数，无论使用复合还是继承，在创建一个Cylinder类的对象时，若不使用初始化列表将会有语法错误。

程序14-3　通过继承重用代码的例子

```
#include <iostream>
using namespace std;

class Circle                                    // original code for reuse
{ protected:                                    // inheritance is one of the options
    double radius;                              // internal data
  public:
    static const double PI;                     // it must be initialized

  public:
    Circle (double r)                           // conversion constructor
    { radius = r; }

    double getLength () const                   // compute circumference
    { return 2 * PI * radius; }

    double getArea () const                     // compute area
```

```
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                          // change size

  const double Circle::PI = 3.1415926536;

class Cylinder : public Circle                  // new class Cylinder
{ protected:
    double height;                              // other data is in Circle

  public:
    Cylinder (double r, double h)               // from Circle plus new code
        : Circle(r)                             // initializer list (no PI)
    { height = h; }                             // new code

    double getVolume() const                    // no getArea()
    { return height * getArea(); }              // additional capability
   } ;

int main()
{
 Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);         // initialize data
 double length = cyl1.getLength();              // similar to Circle
 cyl1.set(3.0);
 double diam = 2 * cyl1.getRadius();            // no call to getArea()
 double vol = cyl2.getVolume();                 // not in Circle
 cout << " Circumference of first cylinder: " << length << endl;
 cout << " Volume of the second cylinder:   " << vol << endl;
 cout << " Diameter of the first cylinder:  " << diam << endl;
 return 0;
  }
```

因此，使用继承设计不会比使用类复合设计难，或者两者一样复杂（例如，初始化列表），或者继承设计更为简单（不需要单线数据成员）。这意味着任何情况下使用继承都比使用复合好吗？当然不是。

使用继承的主要问题是客户代码中没有描述服务器类所提供的服务的代码段。在程序14-2中使用了复合，Cylinder类定义本身就是描述服务器类服务的代码。而在程序14-3中使用了继承，Cylinder类定义只描述了基类Circle所没有的功能，其他可供Cylinder类的客户代码使用的服务是在Circle类定义中描述的。因此，Cylinder类的客户代码的程序员还要了解由基类所提供的服务。当继承层次比较多时，这个问题将更加突出。

这对于C++编译程序不成问题。编译程序搜索继承树来确认客户代码中的消息合法性。设计人员（和维护人员）也需要这样做，但是他们这样做比编译程序要困难许多。

使用继承的第二个问题是客户代码要非常熟悉基类所提供的服务，才能很好地使用派生类并不支持的基本服务。例如，客户代码可能会按如下方式计算圆柱的表面积：

```
double area = cyl1.getArea();    // nonsense - this is not the area!
```

表面上，该函数调用与调用getLength()、getRadius()和set()方法相同。既

然这些方法对Circle类和Cylinder类都是一样的，为什么getArea()就应该不同呢？这些调用对于编译程序来说是一样的，对于Cylinder客户代码的设计人员来说也一样，甚至对于那些不太擅长几何本质的维护人员来说可能也是一样的。

因此，就要由Cylinder类的设计者来保证，Cylinder的客户代码可以使用getLength()、getRadius()和set()方法，但不能使用getArea()方法。如何才能实现这种设计呢？其中一种办法是使用private继承或protected继承。

```
class Cylinder : protected Circle       // new class Cylinder
{ protected:
    double height;                      // other data is in Circle
  public:
    Cylinder (double r, double h)       // from Circle plus new code
        : Circle(r)                     // initializer list
    { height = h; }                     // new code
    double getVolume() const            // no getArea()
    { return height * getArea(); }      // additional capability
  } ;
```

但是这样做又太过分了。的确，Cylinder类的客户代码将不能调用getArea()，因为它是Cylinder类的protected成员。但同时，getLength()、getRadius()和set()方法也成为protected成员。这里有两种补救措施，一种办法是在派生类Cylinder类中显式定义getLength()、getRadius()和set()方法为public成员，而不对getArea()方法或者其他应该在派生类中隐藏掉的基类的服务进行这样的处理。

```
class Cylinder : protected Circle       // new class Cylinder
{ protected:
    double height;                      // other data is in Circle
public:
      Circle::getLength;                // no getArea() here
      Circle::getRadius;
      Circle::set;
  public:
    Cylinder (double r, double h)       // from Circle plus new code
        : Circle(r)                     // initializer list (no PI)
    { height = h; }                     // new code
    double getVolume() const            // no getArea()
    { return height * getArea(); }      // additional capability
  } ;
```

这种方法并不像看上去的那么糟糕。在派生类中将public基类方法定义为protected，然后再将某些基类方法定义为public，这样的做法的确有些难看。但从软件工程的角度而言，这样非常好。为什么呢？因为Cylinder类使用显式的公共成员列表将服务提供给客户，这是一个代码自我文档化的好例子。

另一种补救办法如程序14-4所示，派生类Cylinder将继承来的方法显式地提供给客户代码使用，但那些客户代码不应该访问的方法除外（如getArea()）。这与使用类复合的程序14-2非常相似。注意在这些单线函数中类作用域运算符的使用。在类复合中，是把消息传递给数据成员的；在继承中，没有显式的数据成员，只有基类对象的基本部分。漏掉这些作用域运算符将会导致无穷的递归调用。

```
void Cylinder::set(double r)
```

```
{ set(r); }                          // implicit recursive call
```

这个set()函数在Cylinder类的作用域内调用，根据函数调用解析规则，编译程序首先查找属于局部作用域的名字。由于这个名字在Cylinder类中，编译程序将再次调用Cylinder::set()，而不去根据继承链调用Circle::set()方法。下面的代码等价于上面的代码。

```
void Cylinder::set(double r)
 { Cylinder::set(r); }               // explicit recursive call
```

为了避免无穷递归调用，使用Circle作用域运算符是必要的。

程序14-4 通过受保护继承重用代码的例子

```
#include <iostream>
using namespace std;

class Circle                                  // original code for reuse
{ protected:                                  // inheritance is one of the options
    double radius;                            // internal data
  public:
    static const double PI;                   // it must be initialized

  public:
    Circle (double r)                         // conversion constructor
    { radius = r; }

    double getLength () const                 // compute circumference
    { return 2 * PI * radius; }

    double getArea () const                   // compute area
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                        // change size

  const double Circle::PI = 3.1415926536;
class Cylinder : protected Circle                 // new class Cylinder
{ protected:
    double height;                                // other data is in Circle

  public:
    Cylinder (double r, double h)                 // from Circle plus new code
        : Circle(r)                               // initializer list (no PI)
    { height = h; }                               // new code

    double getLength () const
    { return Circle::getLength(); }               // from class Circle

    double getRadius() const                      // from class Circle
    { return Circle::getRadius(); }

    void set(double r)                            // from class Circle
    { Circle::set(r); }

    double getVolume() const                      // no getArea()
```

```
    { return height * getArea(); }                    // additional capability
   } ;

int main()
{
 Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);               // initialize data
 double length = cyl1.getLength();                    // similar to Circle
 cyl1.set(3.0);
 double diam = 2 * cyl1.getRadius();                  // no call to getArea()
 double vol = cyl2.getVolume();                       // not in Circle
 cout << " Circumference of first cylinder: " << length << endl;
 cout << " Volume of the second cylinder:   " << vol << endl;
 cout << " Diameter of the first cylinder:  " << diam << endl;
 return 0;
  }
```

该方法弥补了使用继承所带来的两个不足。Cylinder类将其服务显式地列举出来，供客户代码使用。客户代码也不会调用不适合派生类使用的基类中的服务。另一方面，使用程序14-2中的复合也不会有这两个不足。受保护的继承和复合十分相似，但是如果可以选择，我们建议使用复合而不使用受保护的继承，因为复合在概念上要简单一些，而且类之间的链接也不像使用受保护继承那么紧密。

14.1.5 以重新定义函数的方式继承

只有不恰当地使用继承时，才需要在派生类对象中抑制某些基类方法。同时，这也表明派生类的对象不是基类的对象，而是将基类看做数据成员。因此，我们建议使用复合而不使用受保护继承。

通常，类之间的关系与继承关系非常类似，不需要抑制基类中的方法。但在派生类中可以按不同的方式处理基类的方法。getArea()方法就是一个很好的例子。对于基类Circle的一个对象，该方法应返回圆的面积。

```
double Circle::getArea () const              // compute circle area
{ return PI * radius * radius; }
```

对于派生类Cylinder的对象，该方法应返回圆柱中两个圆的面积与其侧面的面积之和。

```
double Cylinder::getArea () const            // compute Cylinder area
{ return 2 * Circle::PI * radius * (radius + height); }
```

当派生类方法隐藏基类方法时，派生类方法可以在基类方法的基础上增加另外一些工作。许多C++程序员喜欢“文档化”这个事实，他们在派生类的方法中显式调用基类的方法（显式地使用基类作用域运算符）。

```
double Cylinder::getArea () const            // compute Cylinder area
{ double area = Circle::getArea();
  return 2 * (area + Circle::PI * radius * height); }
```

在派生类中重定义基类的方法是一种常用的编程技巧。其根本是鼓励C++程序员使用统一的名字。当我们使用COBOL编程时，主管会告诉我们为不同的函数（或段落）使用不同的名字，例如COMPUTE-CIRCLE-AREA和COMPUTE-CYLINDER-AREA。此外，还要求我们使用一个精致的数字前缀系统，以标明每个名字属于程序的哪个单元。

在C++中是不赞成使用这种做法的。我们不确定使用统一命名（例如getArea()）是

否只是出于编程美观性的考虑。它的技术原理应该是，使用名字解析规则确定应该调用哪个方法（基类方法或派生类的方法）具有灵活性。正如前一章所提及的，这些原则最初容易让人混淆，但很快会成为编程直觉的一部分。

重定义的继承通常是公共的，这样比公共继承而不重新定义方式要处理更多的事情：重用了某些设计（如radius和getLength()），增加进来一些新成员（如height与getVolume()），重新定义了某些方法（如getArea()）。程序14-5的设计使用了继承并重新定义了方法。我们对Cylinder类的客户代码作了轻微的改变，以便演示客户代码对getArea()方法的使用。程序的输出结果如图14-2所示。

程序14-5 通过公共继承并重新定义方法的代码重用的例子

```
#include <iostream>
using namespace std;

class Circle                                    // original code for reuse
{ protected:                                    // inheritance is one of the options
    double radius;                              // internal data
  public:
    static const double PI;                     // it must be initialized

  public:
    Circle (double r)                           // conversion constructor
    { radius = r; }

    double getLength () const                   // compute circumference
    { return 2 * PI * radius; }

    double getArea () const                     // compute area
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                          // change size

  const double Circle::PI = 3.1415926536;

class Cylinder : public Circle                  // is Cylinder really a Circle?
 { protected:
     double height;                             // other data is in Circle
  public:
    Cylinder (double r, double h)                       // from Circle plus new code
        : Circle(r)                                     // initializer list (no PI)
    { height = h; }                                     // new code

    double getArea () const                             // compute Cylinder area
    { return 2 * Circle::PI * radius * (radius + height); }

    double getVolume() const    { return height * getArea(); }
                                                        // additional capability
   } ;

int main()
{
```

```
Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);              // initialize data
double length = cyl1.getLength();                   , similar to Circle
cyl1.set(3.0);
double diam = 2 * cyl1.getRadius();double vol = cyl2.getVolume();
                                                    // not in Circle
cout << " Circumference of first cylinder: " << length << endl;
cout << " Volume of the second cylinder:    " << vol << endl;
cout << " Diameter of the first cylinder:   " << diam << endl;
cout << " Area of first cylinder: " << cyl1.getArea() << endl;
return 0;
}
```

```
Circumference of first cylinder: 15.708
Volume of the second cylinder:   2945.24
Diameter of the first cylinder:  6
Area of first cylinder: 169.646
```

图14-2　程序14-5的输出

当派生类重新定义基类的方法时，仍使用相同的方法名。在这个例子中，基类和派生类方法的接口也相同。当基类和派生类的函数都完成相似的操作时，它们的方法名可以相同。尽管操作的对象可能不同，但总的语义（意义）是相同的。因此接口相同是很自然的。

在C++中，允许重写一个基类函数并保持其函数接口不变，但这不是必须的。不论出于何种原因，许多C++程序员认为接口应该保持相同。但实际上并非如此。无论派生类改不改变方法的接口，基类方法名对于派生类的客户代码总是隐藏的，客户代码调用的是派生类的方法。如果需要的话，客户代码也可以使用基类方法，但这要求使用基类作用域运算符，让编译程序和读者都知道使用的是基类方法。

```
Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);                  // initialize data
  double length = cyl1.getLength();                     // similar to Circle
  cyl1.set(3.0);
  double diam = 2 * cyl1.getRadius();double vol = cyl2.getVolume();
                                                        // not in Circle
  cout << " Circumference of first cylinder: " << length << endl;
  cout << " Volume of the second cylinder:    " << vol << endl;
  cout << " Diameter of the first cylinder:   " << diam << endl;
  cout << " Side area of first cylinder: "
       << cyl1.Circle::getArea() << endl;                // visual cue
```

14.1.6　继承和复合的优缺点

继承是一种很好的抽象工具，它显式地强调了类之间所存在的概念性联系。例如，在程序中通过从Circle类派生Cylinder类，使Circle类和Cylinder类之间的共性得到了最好的体现。在Cylinder类的代码中继承关系非常明显。客户端代码程序员和维护人员不需要单独研究这些类，不需要通过比较几个类的代码就可以知道哪些类是相似的。

使用继承可以帮助我们节省开发时间，这并不意味着它能使实现类的源代码更短些，情况常常恰好相反。但仍有许多程序员认为，以简单的步骤开发一个复杂的类，比开发一个单独而庞大的单元要简单。例如，开发Circle类让设计人员先将注意力集中到相对简单的操作上（如计算周长），然后当Circle类已经设计好后，再处理比较复杂的任务（如计算圆柱

体积或面积）。

但是，使用继承带来了类之间额外的隐式依赖。这种依赖对于客户代码的设计人员和维护人员而言可能不明显。在派生类描述中也没有提供服务列表，因而还要查看基类的描述。

使用类复合对于继承而言是个很好的替代品。聚集类提供了所支持的所有服务的列表。类复合也带来了类之间的依赖关系，但这些依赖是显式的，以单线方法的形式出现。它将任务从容器类转移到组件类。

选择使用继承还是使用复合依赖于相互关联的类之间的相似程度。如果它们共同的方法比较少而且要增加大量的额外服务，使用复合较好。聚集类只用编写几个单线函数，而且写这些函数的开销相对显式提供有效服务的列表的重要性而言微不足道。

当两个类公共的方法很多而且要增加的服务较少时，使用继承。许多程序员不愿编写“哑”单线方法，使用继承可以去掉这些方法——基类的方法将直接被派生类所继承。在派生类中重新定义基类方法也十分方便，而且为多态性（在下一章中讨论）的使用提供了可能性。

在使用继承时，尽可能使用最简单的方式，如公共继承。避免使用受保护继承和私有继承。

继承常常只是用来加快开发程序的速度，而没有清楚地表达类之间的“是一个”联系。如果怀疑类之间并不存在自然的“是一个”联系，就不要使用继承，最好使用复合。

14.2 统一建模语言

在编写传统程序时，将程序看作相互协作的函数组成的系统。而编写面向对象的程序时，则看作是相互协作的类组成的系统。而类包含有数据和函数。本书第一部分，我们主要介绍了函数的编写方法。为函数编写处理代码以及实现函数之间通信的技巧，对于高质量的C++程序是至关重要的。

本书第二部分主要介绍了类的编写方法。将数据和操作绑定在一起以及将类作为服务器或客户来实现的技巧，对于高质量的C++程序也是至关重要的。

本部分我们将主要讨论相互关联的类的编写方法。我们将会介绍实现类之间关系的方法，如继承、复合等，它们对于高质量的C++程序同样是至关重要的。

当C++程序非常复杂时，不仅很难看出程序中已实现的类之间的关系，而且，对于将要实现的类也很难看出它们之间的关系。

14.2.1 使用UML的目的

解决上述问题的一个常用办法是使用图形符号来描述现实生活中实体之间的关系，应用程序将要模拟它们的行为，如圆与圆柱、顾客与账户、库存项目与供应商等。这是面向对象分析的任务，要以类之间协作的方式描述系统行为，而不是以操作（函数）之间协作的方法进行描述。

下一步骤是决定哪些现实生活中的实体应表示为C++程序中的类，以及决定哪些实体关系可由C++程序中类之间的关系来实现，这些都是面向对象设计的任务。面向对象设计也使用图形符号来描述程序中的类与类之间的关系。

面向对象的系统分析主要描述程序（与用户和其他系统的）外部界面，而面向对象设计主要描述系统的结构元素，包括：系统体系结构、子系统在不同硬件组件之间的分配，不同

类之间的链接关系等。大多数链接关系与在面向对象分析阶段所建立的实体之间的链接关系是相同的，只是分析阶段的某些链接关系可能不在程序中实现，也可能要增加类（和一些附加类）之间的某些其他链接关系来提高系统的性能，改善用户界面。

在本书的前两部分，我们讨论了减少系统组件之间的依赖性的方法。这些方法非常有用，因为系统组件之间的依赖性要求开发人员之间进行合作，而这种合作常常容易出错。但如果要求系统组件之间完全相互独立，又是不现实的。由于这些组件属于同一系统，它们不可避免地要与其他组件相互协作。重要的是尽可能地减少这些协作，同时，以恰当准确的方式描述这些协作也很重要。

于是引入UML表示符号。在传统的系统开发过程中，分析、设计及实现阶段都使用不同的图形符号表示方法，以满足分析人员、设计人员和程序员们的不同要求。在面向对象的系统开发过程中，分析人员、设计人员和程序员们都使用相同的符号表示方法。与传统方法相比，它有两个突出的优点：首先，合作的开发人员（分析人员、设计人员和程序员）使用相同的图形符号表示方法，可以减少误解和对隐式假设的不同解释。其次，在开发的各个阶段，其表示形式没有大的改变，这样可以减少在各阶段之间的转换过程中潜在的错误。

UML是一种强大的统一建模语言，它允许开发者使用图形符号来表示系统协作组件之间的关系。“协作”指的是系统组件之间互相了解、互相使用、互相依赖。这种图形符号表示的基础是将对象作为一个系统组件。对象的概念涉及到数据成员、成员函数以及对象之间的关系。这些对象图可由系统用户和系统实现者进行讨论，并确认是否正确地反映了对象之间的关系。稍后，这些图可转化为面向对象的实现。

这与我们在前面章节所讨论的面向对象程序设计方法相比，在概念上有重大的飞越。面向对象程序设计的本质是将数据和操作绑定在一起，正是特别的数据成员及其上的操作结合起来表示了一个C++类的特点，类的定义中不涉及关系。这样定义不太好，因为它造成了错觉，让人误以为将数据和行为结合起来就完全足够描述对象。

面向对象分析和面向对象设计采用不同的步骤。它们将对象描述成数据、行为及其与程序其他组件之间关系的结合。正如我们将要看到的，在UML中，对数据和行为的描述是非常基础的工作，而类与对象之间的关系才是大多数UML符号描述的重点。

面向对象编程的方法没有强调关系或关联的重要性，因为程序员要尽可能地使程序组件之间相互独立。侧重于组件独立性的结果是，所有的面向对象语言都给程序员提供了描述数据成员和成员函数的方法，但没有提供描述程序组件之间关系的内在（native）方法。

而实际生活中，程序组件之间常常是相互关联的。这样，程序员只能使用特别的方法描述这些关系：如程序员定义类型的数据成员，指向其他对象的指针或引用数据成员。

每一种设计表示符号——包括UML——的主要任务都是帮助程序员描述对象之间的关系。在实现一个C++程序时，由程序员决定哪些对象之间有关系。如果按设想实现程序不是特别复杂就太好了。使用UML表示方法，开发人员可以比较不同的设计方案，然后选择满足以下两点的关系：1) 这些关系足以处理所要完成的工作，2) 使程序对象之间关系的复杂度最小。

UML将三种不同的符号表示方法结合起来建立计算机系统的图形模型。这些模型有助于开发者分析系统的需求。系统需求通常描述了系统功能、用户界面、与其他系统的接口、性能以及可靠性等。图形符号表示方法以系统组件之间的关系来描述这些需求。当然这是一个富于挑战的工作。

其中一种符号表示系统是由Grady Booch和他的Rational软件公司所开发的，称为Booch方法。这种符号表示包括系统的几种视图（view），每种视图都有自己的图表（diagram）。Booch图中的对象符号是不规则的“云”状（与我们在前面几章中使用的客户-服务器图很不相同），因此手工很难绘制。Booch是最早意识到图形建模需要基于计算机的工具支持的人之一。Rational公司所开发的Rational Rose是用于面向对象建模最成功的工具之一，该工具对Booch方法的普及也起了很大作用。随着UML的出现，Rational Rose也进行了修改以支持UML的符号表示。

另一种符号表示系统是由James Rumbaugh和他在通用电气公司的协会所开发的，称为对象建模方法（Object Modeling Technique, OMT），除了描述系统中对象之间关系的对象模型之外，OMT还包括另外两种模型：动态模型（dynamic model）和功能模型（functional model）。这两种模型不是真正的面向对象模型，但它们表示了两种著名的设计分析工具：状态转换图和数据流图。这种合成方法可以使那些对状态转换图和数据流图很熟悉的开发者平滑地转移到面向对象方法。也许正因为此，OMT在业界已被认为是标准的开发方法，获得广泛的应用。

第三个符号表示系统是瑞典的Ivar Jacobson和他的公司开发的目标系统（Objective System），以面向对象软件工程（Object-Oriented Software Engineering, OOSE）和Objectory的名义将该符号表示系统推向市场。它用use case描述系统与外部角色之间的交互关系，外部角色包括系统操作者和其他系统的接口。

每一种符号表示系统都介绍了如何将这些符号应用在以下阶段：首先是面向对象分析，然后是面向对象设计，最后是面向对象实现。它们都极力解释为什么面向对象方法优于传统的方法，但坦白来说这些解释都不是特别清楚。对于已经相信面向对象方法优点的人来说，这些表示系统很适用。而对于那些不明白为什么面向对象方法会节省开支、提高质量的人来说，这些表示系统不会给他们留下深刻印象。

然而，由于传统的系统开发方法存在着很大的缺陷，业界愿意接受任何能改进目前事务状态的方法。那么接受哪种方法呢？除了由Booch、Rumbaugh与Jacobson所开发的符号表示系统外，还有许多类似的表示系统，如Shlaer&Mellor、Yourdon&Coad、Coleman等。所有这些符号表示系统都很相似，它们都是在P.chen于1976年所提出的用于数据库设计的实体-关系图的基础上修改或扩充而来。

许多年来，各个符号表示系统的开发者们都致力于向公众论证：哪种系统更好以及为什么好。这些论证的基本思想是，对方法进行选择是很重要的决定，因此必须十分小心。一些专家认为，一种方法对于某种软件系统（如实时系统）而言比其他方法要好，而对于另外一种软件系统（如商务系统），该方法可能不如其他方法。另外一些专家认为，在某一阶段一种方法可能比其他所有方法好。称这些争论为“方法战”，但是实际上互相竞争的方法之间的不同在于表示符号，而不是方法论，而表示符号之间的不同又没什么特别影响。

大约在1995年，Booch、Rumbaugh和Jacobson决定创建一种统一的符号表示系统，使它成为主要的建模语言。（另外一种说法是，Booch雇用了Rumbaugh和Jacobson为Rational工作）。UML就是他们合作的结晶。他们写了许多介绍UML及其使用方法的书，Rational Rose工具也完全支持了UML符号表示。

不足的是，由于UML结合了多种不同的思想，因此很复杂。描述UML的书也很难懂。学习该建模语言会有一定的难度，因为有很多的内容需要学习。而且如果一个设计人员做出的

建模决策不好，并没有编译程序来标识这个错误（不像C++编译程序可以在我们出错时帮助我们）。因此，学习UML的过程要比学习C++的过程缓慢很多。

幸运的是，我们不需要成为UML的专家，就能够对程序中对象关系结构进行交流。在本书中，我们只介绍UML的基本知识，这已经足够讨论C++程序中的对象关系了。

14.2.2 UML基础：类的表示

在UML中，对象是类的实例，类是对象类型的描述。类描述了某种对象类型的属性和行为。包括在UML模型中的类主要来源于对问题域中的概念和实体的分析。例如，对于一个商务系统而言，其模型中的类有Customer、Item、Shipment、Requisition、Invoice等；对于一个实时系统而言，其模型中的类有Sensor、Display、CustomerCard、Button、Motor、Lock等。

模型中的类可以用类图来表示。一个类可用一个矩形表示，该矩形分为三个部分：最上面的部分包含类名，中间部分包含类属性，最下面的部分列举了操作。在C++中实现类时，属性成为数据成员或域，操作成为成员函数或方法。图14-3a是UML中类的通用表示图。图14-3b所对应的是Point类的特例，它有x和y两个属性，有set()、get()、move()和赋值运算符等操作。图14-3c所对应的是Rectangle类，其属性有thickness、pt1和pt2，操作有move()和pointIn()。

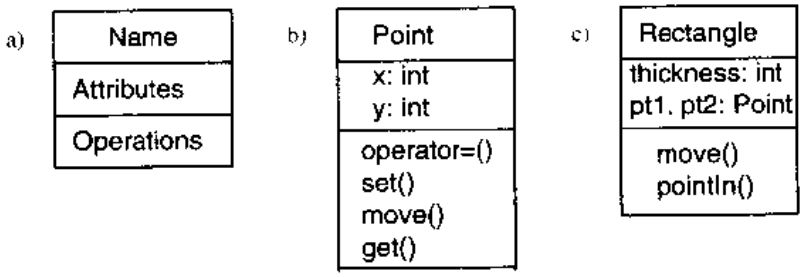


图14-3　普通类的样板和两个具体类的UML示例

在UML中，可以将属性的类型定义为原始（基本）数据类型或者类库中的类型（如String）或应用系统中定义的类（如Point）。UML还允许我们指定除属性名字和类型以外的更多信息。我们可以标明属性是否为静态（类作用域属性）、属性允许的取值集合（如果此属性是枚举类型）、属性的初始值（如果有初始值的话），甚至还可以指定属性的可见性（公共、私有或受保护）。这些信息都是可选的，因为在分析设计的开始阶段，开发者常常还不能确定这些属性的类型或其他性质。可能需要在优化设计的迭代过程中或甚至在编程阶段才能确定这些性质。

对于操作，UML允许定义操作的标识（signature）：名字、返回值、参数的名字和类型。如果需要，还可以指定缺省参数值，也可以定义操作是否为静态操作（类作用域操作），以及定义操作的可见性（公共、私有或受保护）。

正如我们看到的，UML对类的描述细节可以与C++中的类定义非常相似。由于属性和操作的UML符号表示对于所讨论类之间的关系作用不大，我们将不作重点介绍。而且，为了使类图便于管理，开发者常常省略类符号表示中的操作部分，而使用类图讨论类之间的关系。这样，类图中只有两部分，即类名和属性名。对于非常复杂的类图（大多数类图都是复杂的），甚至可以省略其中的属性部分，而直接用一个表示类名的矩形来代表类。这是最便于讨论类

之间的关系的方法。

14.2.3 UML基础：关系的表示

应用程序域中的实体在现实生活中可能是相互关联的，它们之间的这种联系在类图中用类之间的联系表示。描述类之间联系的技术术语是关联（association）。类之间存在着关联表明这两个类的对象之间有链接（link）。这种链接可能是指一个对象知道另一个对象，或者指一个对象连接（connect）到另一个对象上，或者指一个对象使用另一个对象以实现其目的，甚至指对于一个类的每个对象都存在另一个类上有个对象与之对应。常用而且重要的是，在C++程序中实现这种关联以通过一个对象访问另一个对象。

图14-4是关联的例子，图14-4a表示了每个Circle对象与一个Cylinder对象关联，但没有定义关联的性质。注意在表示类的矩形里只有类名，没有属性和操作。它们也可以放在矩形中，但只会使类图更加凌乱。只有当属性和操作有助于明确对象之间的关系时，才将它们放在类图中。

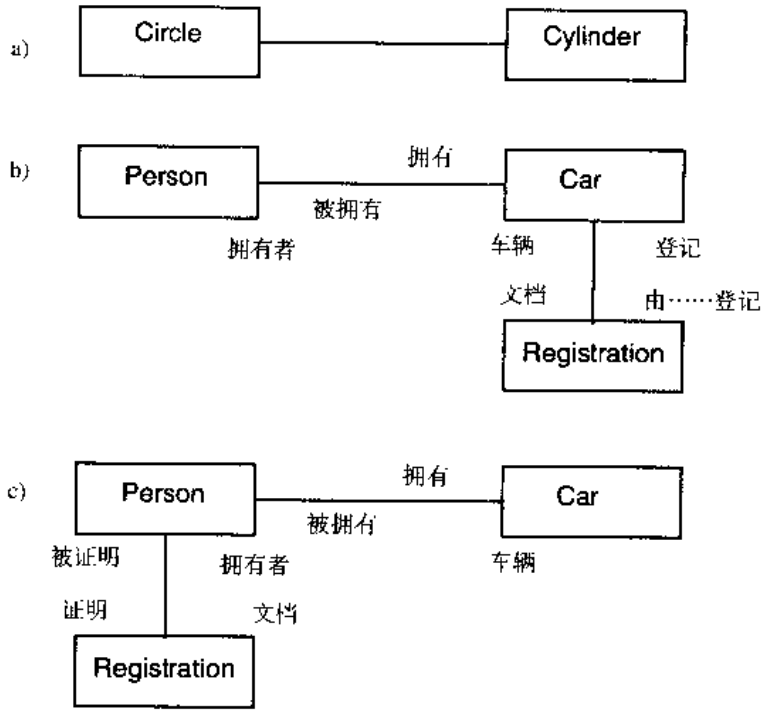


图14-4 类之间关联的UML示例

对象之间的关联通常是双向的。如果一个Circle对象与一个Cylinder对象相关联，那么反过来，该Cylinder对象与Circle对象也是相互关联的。我们不需要再对这两者之间的关联进行说明了。但是UML允许我们提供关联的其他信息，可以给关系命名，也可以给相互关联的对象分配角色。

图14-4b表示一个Person对象可以与一个Car对象相关，一个Car对象可以与一个Person对象和一个Registration对象相关。每个关系有两个标签，分别表示两个方向的关联。为了避免混淆标签与不同方向的关系，可在标签后用一个小箭头标识方向。

在图14-4a中，很难找到适合Circle和Cylinder对象之间关联的好名字。它们只是相

关而已。图14-4b中，我们可以说一个Person对象拥有一个Car对象，而Car对象被Person对象所拥有；还可以说Car对象被Registration对象所登记，而Registration对象登记了Car对象。我们还为关系中的每个对象分配了角色：Person对象是拥有者，Car对象是车辆，Registration对象是文档。

描述关系之间的关系可能并不是很复杂的事情。实际上，只知道对象之间的联系名字及对象所扮演的角色，还不能帮助我们理解在现实生活中和程序执行过程中对象的合作关系。但是，如果开发者对应用程序域知之甚少，关联的名字和对象的角色可能有助于以正确的方式进行分析。

通常，为一个类模型中的关系比较不同的命名看起来像是用简单术语描述相对论。描述关联的主要问题是任何选择都是相对的。图14-4c中使用了不同的关系来描述Person、Car和Registration类之间的关联。还有第三种描述的方法，即每个类都和其他两个类有关联。哪一组关系更好呢？这个问题没有绝对的答案。关系的选择是相对的。

关系在C++中可用指针（或引用）来实现，从一个对象指向另一个相关联的对象。另外还有一种很常用的实现方法是将一个对象的标识符作为另一个类的属性。例如，Person类有一个属性可以标识与Person实例相关联的Car对象。如果需要，在Car类中可将Person的标识符作为一个属性，该属性将Car实例与Person实例关联起来。

14.2.4 UML基础：聚集和泛化的表示

聚集是关联的一种特例，它表示两个类通过一种特殊关联而相关。这种关联表示的是整体-部分关系，即一个对象是另一个对象的一部分，或者说，一个对象包含另一个对象。

UML表示聚集的符号与表示关联的符号相同：都是通过类之间的链接来表示的。要表示聚集对象，只用把一个空心菱形放在链接线与聚集对象相连的一端。显然这个菱形只能放在链接线的一端而不是两端。

图14-5a表示Circle对象是Cylinder对象的一部分。实际上，空心菱形表示该聚集是可共享的，在同一时刻，“部分”对象可能参与了不止一个聚集关系。在复合聚集中不允许共享。UML中复合聚集的表示与共享聚集大致相同，只是复合聚集对象端的菱形是实心的而不是空心的。图14-5b说明了复合聚集的表示，一个Circle对象只能是一个Cylinder对象的一部分，而不能是其他对象的一部分。

既然聚集是关联的特例，因此经常可以把对象之间的关系表示为关联，而不是（共享或复合）聚集。例如在图14-4a中，关联用来对Circle和Cylinder对象之间的关系建模。但是，这种模型不够精确。设计人员的任务是把聚集表示为聚集而不是关联，这样实现时会简单一些。当然，如果聚集不能很好地表达对象之间的关系，就应该使用关联。设计人员常常很苦恼，需要在使用一般的关联还是特殊的聚集之间进行权衡。

共享聚集在C++中的实现方法也与关联的实现方法相似，使用指向组件对象的指针（或引用）。复合聚集实现时，将部分对象作为聚集对象的数据成员。

泛化描述的是一般类和特殊类之间的关系。特殊类除了包含有一般类中的属性和操作，还可能包含其他属性和操作。泛化在面向对象程序设计语言中用继承来实现。泛化表示类之间的“是一个”关系。注意这里指的是类之间的关系，而不是对象实例之间的关系。一个类可以继承另一个类，但一个对象实例不能继承另一个对象。

泛化关系中的特殊类（子类）继承一般类（超类）中的所有属性、操作和关联。在UML

中用类图中的一条实线表示这种关系。为了区分关系中的子类和超类，在超类和链接直线之间插入一个小的空心三角形，并使该三角形指向超类。

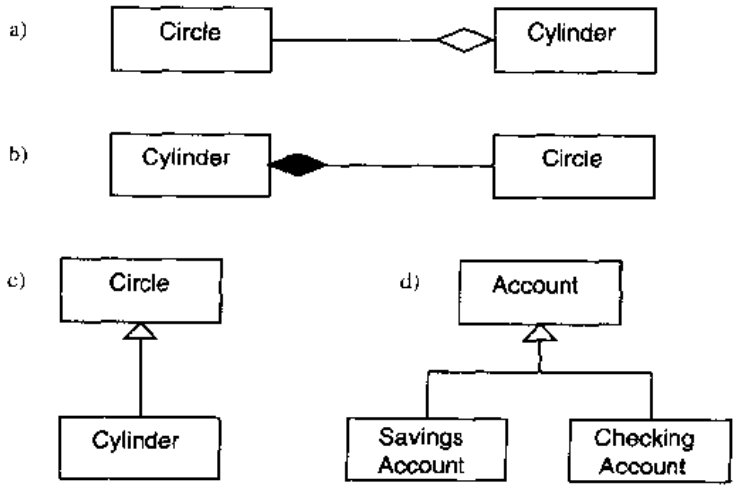


图14-5 共享聚集、复合聚集及继承的UML示例

图14-5c表示了Circle和Cylinder类之间的泛化关系。在此设计中，Circle作为一般类（超类），Cylinder作为特殊类（子类）。

如果一个类在多个泛化关系中充当超类，通常在类图中分别表示这些类，然后每个特殊类都通过带有各自三角形的链接线链接到超类。更加常见的用法是，只使用一个三角形指向超类，而每个特殊类都与该三角形相链接。图14-5d中Account类是一般类，SavingsAccount和CheckingAccount代表Account类的两个特殊类，分别表示了与Account类的不同泛化关系。

如果一个子类又作为另一个类的一般类，即另一个类为该子类的特殊类，仍可以用上面的方法表示。从一个类继承来的类同时又可以是另一个类的基类，这样就产生UML类图中的继承树层次结构。

14.2.5 UML基础：多重性的表示

大多数关系是二元关系——它们链接两个类。实际上还有其他情况，例如前面讨论的Person、Car与Registration三个类之间的关系，这是一种三元关系：涉及到三个类的对象。将三元关系用一组二元关系表示是比较困难的。

即使UML能够支持三元关系的表示符号，但对多于三个类之间的关系则没有提供表示方法。即使提供了所需的表示方法，在实现这些关系时仍会出现问题，因为C++语言只支持二元关系，即在两个对象之间使用物理的或概念性的指针建立链接。因此，我们在建模UML类图时要使用二元关系链接两个类的对象。

有一种例外情况是，关系的对象是同一个类的实例。例如，一个作为主管的Person类对象和另一个作为工作人员的Person类对象之间存在着关系，这两个对象都属于同一个类。这在理论上有点奇怪，大多数关于UML的书也提供了这种自反的（或递归的）关系。实际上，用Supervisor类对主管建模，用TeamMember类对工作人员建模会更加方便一些。在模型中使用两个不同的类是很有用的，因为它们实现了不同的任务。如果这两个类有许多共同的特点，则可以使用Person类作为它们共同的基类。

因此，UML类图中的大多数关系是二元关系，每个链接连接的是两个不同类的两个对象，分别处于链接线的两端。每条链接线连接的是两个对象而不是三个或四个对象。

有时，一个类的对象可与另一个类的多个对象相关联。例如，Supervisor类的一个对象可能与TeamMember类中的多个对象相关联。在UML类图中，Supervisor类和TeamMember类之间仍只有一个链接，但可以用其他的UML符号表示这种多重性。

图14-6表示了类图中的多重性，图14-6a中演示了Point和Rectangle类，在应用程序中，每个Rectangle对象都正好与两个Point对象相关联。这种应用于关联的UML表示方法同样也适用于聚集。图14-6b中也演示Point和Rectangle类之间关系，但这次是作为聚集而不是一般的关联。

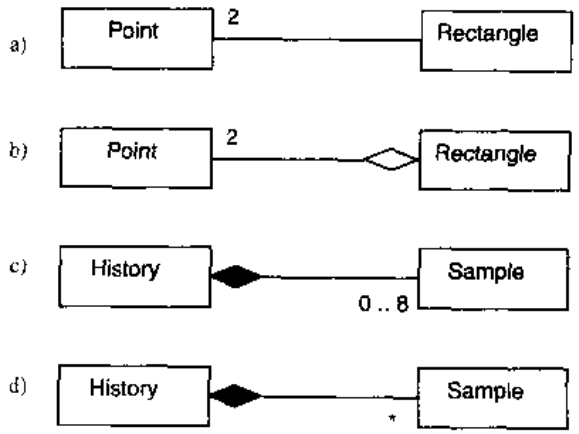


图14-6　表示关系多重性的UML例子

注意，图14-3c表明Rectangle类有两个Point类的属性：pt1和pt2。这意味着Rectangle类的任一对象都正好与Point类的两个对象相关联。因此，图14-6a和14-6b中类之间的链接关系所表达的分析和设计信息与类图完全相同。有些专家认为这样造成了冗余，建议只使用一种表示方法。

比较好的办法是在设计类时只表示关系而省略属性。这样处理基于的原则是：关联反映了分析和设计角度，而属性代表了实现角度。因此在分析角度我们要标明关系，在实现时再用合适的指针、数据成员等实现这些关系。我们尚不确定从实践角度来说这种讨论是否重要，就个人观点而言这很自然。既然不需要关心UML编译程序，我们可以随心所欲。

如果关联或者聚集链接线一端没有其他符号修饰，这就意味着在此关系中该类的对象是操作性的。图14-6a和14-6b显示了必须恰好有一个Rectangle类的对象存在。

有时，对象之间的关系不是固定的，在程序执行过程中其多重性也会发生变化。例如，第12章中的History类与Sample类相关联。实际上，这种关系是一种复合关系：History类的对象包含Sample类对象的数组。如程序12-4所示，在程序开始执行时，数组中没有有效的Sample对象。在执行过程中，测试样本到达，这些信息被存储在数组中直到程序终止或Sample对象个数已达到最大值（8个）。

UML允许指定所关联对象的范围以表示这种可变的多重性。图14-6c中表示所关联对象的个数可在0到8之间变化。如果对象个数不能少于1，则其范围从1而不是0开始，例如1,...,8。

对象的变化范围常常是出于实现的考虑而人为指定的，并不是应用程序域本身决定的。例如，在测试记录中的Sample对象个数只能限制在8以下吗？这只是因为C++不允许不指定

其长度就定义一个对象，因此我们不得不指定一个数字。我们也可以取其他任何数字，在应用程序域中没有任何信息表明8比10、20、100或其他任何数字都要恰当。

从概念上讲，不应该限制测试记录中的样本数目。同样，出于实现的考虑也不应该强制设计人员设置一个特别的数字。程序12-7显示了能动态分配内存的容器类。图14-6d表示了这种无限制的多重性。

14.3 案例分析：一个租赁商店

下面我们讨论一个影碟租赁商店中记录顾客借还影碟的例子。

为了简化，我们假设该租赁商店很小，而计算机内存相当大。应用程序在开始执行时将顾客与租赁项目的数据库装入了计算机内存。租赁项目数据包括项目名、当前数量、项目号。顾客数据包括顾客姓名、电话号码（电话号码作为顾客号）、该顾客已借的影碟数目以及影碟号等。

即使这个例子很小，但足够说明在设计类，建立类之间的关系和为尽量保持类之间独立性而进行设计优化的过程中存在的基本问题。

为了更有意义地使用继承，我们还可以添加以下信息：影碟数据存放在一个文件中，并用一个字符表示该影碟的类别（f表示故事片、c表示喜剧片、h表示恐怖片）。当数据读入内存时，有关影碟类别的信息以数字形式存放（1表示故事片、2表示喜剧片、3表示恐怖片）。当影碟信息显示在屏幕上时，用一个单词来表示其类别（“feature”表示故事片、“comedy”表示喜剧片、“horror”表示恐怖片）。当数据存回到文件中时，再以一个字符来表示类别。

当一个顾客到柜台处租借影碟时，店员输入该顾客的电话号码查找数据库，如果没有找到该顾客的数据，将显示提示信息。如果找到了，该顾客姓名和电话号码及已租赁的影碟数据都显示出来。确认顾客姓名后，店员输入要借的影碟号。该影碟目前的数量减少一个，影碟号被增加到该顾客所借的影碟号列表中。

如果使用条形码光笔输入影碟号，该影碟肯定在数据库中。在这个原型中，当手工输入影碟号时，如果没有找到该影碟，则出现错误消息。

当顾客到柜台处归还影碟时，店员输入该顾客的电话号码，当顾客记录显示出来后，再输入影碟号。如果在该顾客已借影碟列表中找到了该影碟号，则从这个列表中删除该影碟号，该影碟号目前的数量增加一个。如果输入的影碟号不正确，则显示错误信息。

为了简单起见，我们没有考虑费用方面的细节（收租费和超期费等）、程序的性能（对每个影碟的需求的累积提示）及程序管理（加、减、编辑顾客和影碟数据）等细节。

类及其关联

我们常常通过分析功能说明书和其他描述系统用户界面和行为的文档，来编辑一个应用程序的类列表。

一些专家建议列举出系统描述中的所有名词，以此作为进一步工作的开端。另外一些专家则嘲笑这种做法，因为大多数描述实体的名词尚未到达类的级别（例如电话号码、当前数量等），因此这些名字最后只能作为属性（数据成员）而不是类。

使用系统描述建立模型时需要注意的一点是，这个描述重点在于与系统交互的实体上（例如，顾客、店员、数据库）。而建模的目标是最终产生系统实现，该实现由包含数据及其操作的类（例如Customs，StoreClerk和Database）组成。系统描述中的实体和系统实

现中的类可能有相同的名字，但是意义并不相同。

忽略细节，我们可以假设类模型中应包括以下类：Item（有关影碟的信息）、Customer（有关顾客的信息）、Inventory（管理项目和顾客集合）、File（管理库存项目和顾客数据库）。这些类及其属性、操作如图14-7所示。

图14-7中的大部分属性和操作所表示的意思比较明确，那些不太明确的将在进一步讨论中进行解释。

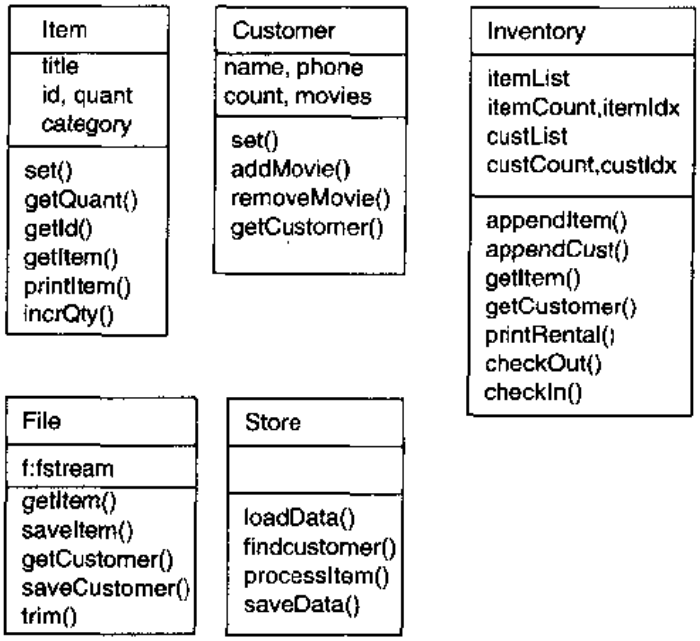


图14-7　类及其属性、操作的UML表示

类与类之间是如何关联的呢？必须承认这个系统描述不太有助于分析出类的关联。但这并不是错误，因为，描述通常是用来帮助建立和测试系统的，而不是辅助绘画系统的UML模型的。

我们认为学习创建类模型的最好方法是先为一个简单的应用程序创建几个可选方案，然后实现每个方案，并从复杂性角度对这些方案进行评估。这个例子的大小正好适合于这种学习过程。

然而，似乎像我们这样的做法比较少。大多数关于面向对象分析和设计的书都认为，在一开始就根据系统描述提供类图的例子比较合适，而没有进一步的实现，最重要的是没有评估模型是如何影响解决方案的复杂性的。但实际上，选择如何在类之间分配属性和操作，以及选择如何用关系将类链接起来，都会影响程序的复杂性、可重用性和可维护性。

显然，Item类和Customer类应是其他类的服务：它们提供存取数据成员值的服务。

在多文件项目中，每个类的定义分别放在一个头文件中。头文件被包含在实现类的客户的源文件中，也就是，源文件使用类名去定义变量或参数。程序14-6是Item类及其数据成员和成员函数的头文件。这个类为影碟名、影碟号、当前数量、种类提供了数据成员。其方法允许客户代码设置Item对象的数据成员值，并允许获取对象号、对象数量等所有4个数据成员。还允许客户代码按所需格式（省略目前数量）打印项目数据，以及增加（或减少）一个当前数量。

程序14-6　Item类的定义（文件item.h）

```
// file item.h
```

```
#ifndef ITEM_H
#define ITEM_H

class Item
{
 protected:
       char title[26];
       int id, quant, category;
 public:
       void set (const char *s, int num, int qty, int type);
       int getQuant() const;
       int getId() const;
       void getItem(char* name, int &num, int& qty, int &type) const;
       void printItem() const;
       void incrQty(int qty);
 } ;

#endif
```

注意条件编译指示符的使用。根据从C中沿袭来的规则，一个头文件只能被程序的源文件包含一次。如果在程序的多个源文件中都包含了同一头文件，该类的定义将随着每个源文件编译。由于每个源文件都可单独进行编译，并分别生成目标文件，这样程序中将有同一类型的多个定义，链接时会很麻烦。当然，如果让链接程序在遇到几个相同的定义结构时丢掉多余的定义会简单很多。因此，每个C++程序员都应将这些条件编译指示符放在每个头文件中。这真是令人遗憾。

程序14-7是Customer类的头文件，同程序14-6一样使用了条件编译指示符，Customer为存储顾客姓名、电话号码、顾客所租影碟数及所租影碟号提供了数据成员。其成员函数允许客户代码设置顾客名与电话号码，将影碟号添加到影碟列表中，从影碟列表中删除一个影碟号，以及获取顾客名、电话号码和顾客所借的影碟列表。

程序14-7 Customer类的定义（文件customer.h）

```
// file customer.h

#ifndef CUSTOMER_H
#define CUSTOMER_H

class Customer
{
  char name[20], phone[15];
  int count;
  int movies[10];
public:
  Customer ();
  void set(const char *nm, const char *ph);
  void addMovie(int id);
  int removeMovie(int id);
  void getCustomer(char *nm, char *ph, int &cnt, int m[]) const;
 } ;

#endif
```

与头文件类似，多文件项目中每个类的C++源代码也分别实现在不同的源文件中。程序14-8是Item类的实现，该文件中必须包含头文件“item.h”，其目的是让编译程序知道作用

域运算符“Item::”所表示的意思。

程序14-8　Item类的实现（文件item.cpp）

```
// file item.cpp

#include <iostream>
using namespace std;
#include "item.h"                          // this is a necessity

void Item::set (const char *s, int num, int qty, int type)
{ strcpy(title,s);  id=num;  quant=qty;  category=type; }

int Item::getQuant() const                 // used by Inventory::checkOut()
{ return quant; }

int Item::getId() const
{ return id; }                             // in printRental(), checkOut(), checkIn()

void Item::getItem(char* name, int &num, int& qty,
        int &type) const                   // used by File::saveItem()
{ strcpy(name,title);  num = id;
 qty = quant;  type = category; }

void Item::printItem() const               // used by printRental()
{ cout.setf(ios::left,ios::adjustfield);
 cout.width(5);  cout << id;               // it knows its print formats
 cout.width(27);  cout << title;
    switch (category) {                    // different item subtypes
      case 1:  cout << "  feature";  break;
      case 2:  cout << "  comedy";  break;
      case 3:  cout << "  horror";  break; }
    cout << endl; }

    void Item::incrQty(int qty)            // used in checkOut(), checkIn()
     { quant += qty; }
```

该实现表明Item类不需要其他任何类的支持。它确实需要一些库函数，因此，一些设计人员在他们的UML图中包含了库组件作为他们的类的服务器类。我们认为这样做是多余的。我们只对程序组件之间的关系感兴趣，而不关心程序使用的库函数和类。

为了方便跟踪类之间的链接关系，我们在程序中使用了注释语句，表明Item类的每个方法在哪里被调用。这样便于维护。注释语句表明Item类是Inventory类和File类的服务器类。

类似地，程序14-9是Customer类的实现文件，头文件“customer.h”包含在该文件中。对于任何实现文件，除了要包含所用到的其他类的头文件，还需要把自己的头文件包含进来。

程序14-9　Customer类的实现（文件coustmer.cpp）

```
// file customer.cpp

#include <iostream>
using namespace std;
#include "customer.h"                              // this is a necessity

Customer::Customer ()
{ count = 0; }
```

```
void Customer::set(const char *nm, const char *ph)
{ strcpy(name,nm); strcpy(phone,ph); }              // in appendCust()

void Customer::addMovie(int id)
{ movies[count++] = id; }                          // in appendCust(), checkOut()

int Customer::removeMovie(int id)                  // used in checkIn()
{ int idx;
  for (idx=0;  idx < count;  idx++)                // find the movie
    if (movies[idx] == id) break;
  if (idx == count) return 0;                      // give up if not found
  while (idx < count - 1)
  { movies[idx] = movies[idx+1];                   // shift tail to the left
    idx++; }
  count--;                                         // decrement movie count
  return 1; }                                      // report success

void Customer::getCustomer(char *nm, char *ph,     // saveData()
  int &cnt, int m[]) const                         // Inventory::getCustomer()
{ strcpy(nm,name); strcpy(ph,phone); cnt = count;
  for (int i=0;  i < count;  i++)
    m[i] = movies[i]; }
```

在“customer.cpp”源代码文件中没有其他的头文件，这表明Customer类没有服务器类，它本身作为别的类的服务器类。每个函数的注释语句表明该函数在哪里用作服务器类，以让客户代码访问顾客的数据和方法。

Customer构造函数将被借的影碟初始化为0，set()方法为顾客姓名和电话号码赋了新值，addMovie()方法将新的影碟号追加到所借影碟列表的末尾。

removeMovie()方法首先检查影碟号是否在顾客所借影碟列表中（如果提供了可靠的数据输入方法，这一步骤就不必要）。如果在列表中找不到，则返回0表示失败；如果找到了，则将该影碟删除，该影碟后面的其他影碟上移一个位置，使有效影碟数减少一个，并返回1表示成功。

注意，这里是影碟数减少一个，而不是数组中的数据值个数减少一个。数组中的数据值个数并没有减少。在移动后最后的两个数组成员有相同的影碟号。因此，我们是说“有效影碟号的个数”，而不是“影碟号的个数”。

不止一个人认为移动算法中的下标限制不太合适。移动算法容易出错，且很难查找错误。可以在移动之前而不是移动之后将影碟数减少一个，这样的算法更简单易懂。

```
int Customer::removeMovie(int id)          // used in checkIn()
{ int idx;
  for (idx=0;  idx < count;  idx++)        // find the movie
    if (movies[idx] == id) break;
  if (idx == count) return 0;
  count--;                                 // decrement movie count
  while (idx < count)                      // more conventional form
  { movies[idx] = movies[idx+1];           // shift tail to the left
    idx++; }
   return 1; }
```

按如下方式编写循环移动更为简洁，即不将加1操作单独放在一行，而是在移动语句中使用递增运算符。

```
while (idx < count)
  movies[idx] = movies[idx++];              // concise but risky
```

赋值在C++里面是一个表达式，C++保证表达式中的运算符按顺序执行计算，但不保证操作数按顺序计算。这是对的，因为表达式中组件的计算顺序是不确定的。如果表达式从左到右执行，上述循环正常工作。如果表达式从右到左执行，则上述循环是错误的。因此使用容易理解的啰嗦一点的代码比使用让维护人员费解的简洁代码要更好一些。

类似于程序14-7的“item.cpp”文件，我们使用注释来说明Customer类的方法。注释表明Customer类是Inventory类的服务器。

下一个要讨论的是Inventory类，程序14-10是Inventory类的头文件。Inventory类的数据成员有项目列表、一个顾客、列表中有效成员个数、访问表成员的下标。其成员函数允许客户代码追加一个影碟到项目表中，追加一个顾客到顾客表中，获取表中当前项目（由itemIdx下标指向），获取表中当前顾客（由custIdx下标指向），打印顾客所借影碟的有关信息，将影碟登记借入与登记还回等。

程序14-10　Inventory类的定义（文件inventory.h）

```
// file inventory.h

#ifndef INVENTORY_H
#define INVENTORY_H
#include "item.h"
#include "customer.h"

class Inventory {
protected:
    enum { MAXM = 5, MAXC = 4 } ;                // just for the prototype
    Item itemList[MAXM];
    Customer custList[MAXC];
    int itemCount, custCount;
    int itemIdx, custIdx;
public:
  Inventory ();
  void appendItem (const char* ttl, int id, int qty, int cat);
  void appendCust (const char* nm, const char* ph,
                   int cnt, const int *m);
  int getItem(Item& item);
  int getCustomer(char* nm, char* ph, int &cnt, int *m);
  void printRental(int id);
  int checkOut(int id);
  void checkIn(int id);
 } ;

#endif
```

由于Inventory类是Item类和Customer类的客户，Inventory头文件中应包含Item类和Customer类的头文件。

有些程序员不能确定类与类之间的依赖性，于是在每个实现文件中都包含了所有的头文件“以防万一”。他们认为安全总比后悔好。

这是不正确的。如果没有包含足够的头文件不会有什么风险，但是，如果包含了过多的头文件就会造成危害。如果包含少了头文件，编译程序会标注出错的行，说明使用了未定义

的名字。但是，如果包含了不需要的头文件，虽然不会看到出错信息，但是给维护人员和客户端代码程序员阅读程序造成困难。

编译程序会忽略冗余的type定义。阅读程序的人只有在查看类代码，并发现这些type名字在类中根本没有使用后，才会忽略它们。这简直就是浪费精力，增加额外负担，而且也容易造成理解错误。

因此，我们同意那些冒险者的做法（特别是因为不包含不需要的头文件没有风险可言）。为以防万一而包含额外的头文件不是一个好的做法。这并不能避免语法错误，反而会困扰程序的读者。

Inventory类的实现见程序14-11，它只包含一个“inventory.h”头文件。虽然该文件中使用了Item类和Customer类的对象，但编译程序能够识别这些类型名。因为Item类和Customer类的头文件已包含在“inventory.h”头文件中，当然也被包含在Inventory类的实现中。

程序14-11 Inventory类的实现（文件inventory.cpp）

```
// file inventory.cpp

#include <iostream>
using namespace std;
#include "inventory.h"                                    // this is a necessity

Inventory::Inventory()
 { itemCount = itemIdx = 0;  custCount = custIdx = 0; }

void Inventory::appendItem (const char* ttl, int id,
                            int qty, int cat)
{ if (itemCount == MAXM)                                  // used in loadData()
    { cout << "\nNo space to insert item"; }
  else
    { itemList[itemCount++].set(ttl,id,qty,cat); } }
void Inventory::appendCust (const char* nm, const char* ph,
                            int cnt, const int *movie)
{ if (custCount == MAXC)                                  // used in loadData()
    { cout << "\nNo space to insert customer";  return; }
  custList[custCount++].set(nm,ph);
  for (int j=0; j < cnt;  j++)
    custList[custCount-1].addMovie(movie[j]); }
int Inventory::getItem(Item &item)                        // used in saveData()
{ if (itemIdx == itemCount)
   {  itemIdx = 0; return 0; }
  item = itemList[itemIdx++];
  return 1; }

int Inventory::getCustomer(char* nm, char* ph, int &cnt, int *m)
{ if (custIdx == custCount)                               // in findCustomer(), saveData()
   {  custIdx = 0; return 0; }
  custList[custIdx++].getCustomer(nm,ph,cnt,m);
  return 1; }

void Inventory::printRental(int id)                       // used in findCustomer()
{ for (itemIdx = 0;  itemIdx < itemCount;  itemIdx++)
   { if (itemList[itemIdx].getId() == id)
       { itemList[itemIdx].printItem(); break; } }
```

```
    itemIdx = 0;}

int Inventory::checkOut(int id)                    // used in processItem()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
    if (itemList[itemIdx].getId() == id) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx = 0; return 0; }
  if (itemList[itemIdx].getQuant() == 0)
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);
  custList[custIdx - 1].addMovie(id);
  itemIdx = custIdx = 0;
  return 2; }

void Inventory::checkIn(int id)                    // used in processItem()
{ if (custList[custIdx - 1].removeMovie(id)==0)
   { cout << " Movie is not found\n";
     itemIdx = custIdx = 0;  return; }
  for (itemIdx = 0;  itemIdx < itemCount;  itemIdx++)
   { if (itemList[itemIdx].getId() == id)
     { itemList[itemIdx].incrQty(1);  break; } }
  itemIdx = custIdx = 0;
  cout << " Movie is returned\n"; }
```

与程序14-6和程序14-8类似，我们也使用了注释语句标明每个方法在服务器代码的哪部分被调用。与Item类和Customer类不同，Inventory类只有一个客户类——Store类。

Inventory类的构造函数对下标、项目和顾客数进行初始化。开始，这两个列表都是空的。appendItem()方法和appendCust()方法很简单：它们首先测试是否有空间（对原型而言这种测试是合适的，如果内存是动态管理的，这样做就多余了），如果有则将成员追加到数组末尾，并使有效成员数加1。

getItem()方法和getCustomer()方法获取数组中给定下标处的对象数据（itemIdx指向Item对象，custIdx指向Customer对象）。有时候获取整个对象，而有时候只获取对象的数据成员的值。因此在第一种情况下，客户代码可以直接得到数据成员的值；而第二种情况下，是Inventory类代表客户代码得到数据成员的值。这种不一致会导致客户代码的不同行为，增加了阅读客户代码的难度。

printRental()方法用影碟号在itemList[]数组中查找影碟。如果找到了，则给对象发送printItem()消息。

checkOut()方法将影碟号作为参数在itemList[]数组中查找项目。如果没有找到，则放弃并返回0。如果找到了但目前的数量为0，则放弃并返回1。如果项目有效，则将该项目的当前数量减1，并将该影碟号增加到顾客所借影碟列表中，返回2。

checkIn()方法也将影碟号作为参数。它通过调用removeMovie()方法查找顾客所借影碟列表中的项目。如果没有找到，checkIn()方法将打印一条消息并退出。如果找到了，则从顾客所借影碟列表中删除该影碟，并搜索itemList[]数组中的该项目，使当前数量加1，打印确认消息。

checkIn()方法和checkOut()方法的界面不一致。checkOut()方法不涉及用户界面对话框，客户代码要分析它所返回的值，根据返回值打印消息，这样将任务推给了客户代码。checkIn()方法负责对错误情况进行分析及提供相应的用户界面，它对客户代码隐

藏了有关的错误细节，而只返回一个空值。

下一个讨论的是File类，它在程序运行之前和之后访问包含项目和顾客数据的物理文件。图14-8是影碟数据输入文件的例子，文件每一行相当于一个项目：影碟名（左对齐）、影碟号、当前数量及种类（字母形式）。

```
Splash                          101  11  c
Birds                           102  22  h
Gone with the wind              103  10  f
```

图14-8 带有影碟数据的简单输入文件

图14-9是顾客数据输入文件的例子，每一顾客占两行：第一行包括顾客姓名和电话号码，第二行包括所借影碟数及所借影碟号列表。

```
Shtern                          353-2566
2 101 102
Shtern                          358-0808
0
Simons                          277-7506
3 102 101 103
```

图14-9 带有客户数据的简单输入文件

输出文件的格式与输入文件格式相同。程序运行后项目输出文件的内容如图14-10所示。与图14-8比较，可以发现顾客归还了一个名为“Splash”的影碟，借走了一个名为“Gone with the Wind”的影碟。

```
Splash                          101    12   c
Birds                           102    22   h
Gone with the wind              103     9   f
```

图14-10 带有影碟数据的简单输出文件

程序运行后顾客文件的内容如图14-11所示。它表明Shtern顾客归还了一个影碟号为101的影碟，并借走了影碟号为103的影碟。

```
Shtern                          353-2566
2   102   103
Shtern                          358-0808
0
Simons                          277-7506
3   102   101   103
```

图14-11 带有客户数据的简单输出文件

程序14-12是File类的定义。File类将可以读写数据的fstream文件对象封装起来。这个类实现了公共方法getItem()和saveItem()，对Item数据进行输入/输出操作。另外还实现了公共方法getCustomer()和saveCustomer()，以处理Customer数据的输入/输出。

程序14-12　File类的定义（文件file.h）

```
// file file.h

#ifndef FILE_H
#define FILE_H
#include "item.h"
#include <fstream>

class File
{ fstream f;
  static void trim(char buffer[]);
  enum { TWIDTH = 27, IWIDTH = 5, QWIDTH = 6,
         NWIDTH = 18, PWIDTH = 16 } ;
public:
  File(const char name[], int mode);
  int getItem(char *ttl, int &id, int &qty, char &type);
  void saveItem(const Item &item);
  int getCustomer(char *name, char *phone, int &count, int *m);
  void saveCustomer(const char *nm, const char *ph,
                    int cnt, int *m);
} ;

#endif
```

程序14-13是File类的实现，其构造函数打开用于读写的物理文件，并通过调用fail()函数测试操作的成功与否。测试操作成功与否的另一种方法是调用is_open()函数，如果成功打开了文件则返回真。

程序14-13　File类的实现（文件file.cpp）

```
// file file.cpp

#include <iostream>
using namespace std;
#include "file.h"                              // this is a necessity

File::File(const char name[], int mode)
{ f.open(name,mode);                           // used in loadData(), saveData()
  if (f.fail())                                // if (f.is_open()) is OK, too
   { cout <<" File is not open\n";  exit(1); } }

int File::getItem(char *ttl, int &id, int &qty, char &type)
{ char buffer[200];                            // in loadData()
  f.get(buffer,TWIDTH);
  trim(buffer);
  strcpy(ttl,buffer);                          // it knows file structure
  f >> id;  f >> qty;  f >> type;  f.getline(buffer,4);
  if (!f) return 0;
  return 1; }

void File::saveItem(const Item &item)          // in saveData()
{ char tt[27]; int id, qty, type;
  item.getItem(tt,id,qty,type);
  f.setf(ios::left,ios::adjustfield);
  f.width(TWIDTH);  f << tt;                   // it knows file format
  f.setf(ios::right,ios::adjustfield);
```

```
  f.width(IWIDTH);  f << id ;
  f.width(QWIDTH);  f << qty;
  switch (type) {                          // different for different subtypes
    case 1:  f << "   f\n";  break;
    case 2:  f << "   c\n";  break;
    case 3:  f << "   h\n";  break; } }

int File::getCustomer(char *name,char *phone,int &count,int *m)
{ char buffer[200];                        // in loadData()
  f.get(buffer,NWIDTH);
  trim(buffer);
  strcpy(name,buffer);
  f >> buffer;  f >> count;                // it knows file structure
  strcpy(phone,buffer);
  for (int i=0;  i < count;  i++)
    f >> m[i];
  f.getline(buffer,2);
  if (!f) return 0;
  return 1; }

void File::saveCustomer(const char *nm, const char *ph,
                        int cnt, int *m)   // in saveData()
{ f.setf(ios::left,ios::adjustfield); f.width(NWIDTH);
  f << nm;
  f.setf(ios::right,ios::adjustfield); f.width(PWIDTH);
  f << ph << endl << cnt;                  // it knows file structure
  for (int i=0;  i < cnt;  i++)
  { f.width(6); f << m[i]; }
  f << endl; }

void File::trim(char buffer[])             // in getItem(), getCustomer()
{ for (int j = strlen(buffer)-1; j>0; j--)
    if (buffer[j]==' '||buffer[j]=='\n')
      buffer[j] = '\0';
    else
      break; }
```

getItem()方法从输入文件把一行数据读入到局部数组buffer[]中，截断其尾部的空格，再把数据拷贝到输出数组tt1[]中，然后再从文件中把数据读取到项目的其他数据成员中：项目号、当前数量、种类。如果正读取的行是物理文件中最后一行的数据，则最后一个getline()方法调用到达文件终止处。这时文件对象为空，getItem()返回0，告诉调用者（Store类）已到了输入数据的末尾。否则，返回1表示还有数据读取。

saveItem()方法将项目数据保存到物理文件中。为了确保表示种类的整数能正确地转换为相应的字母，该方法使用了switch语句。

getCustomer()方法读取顾客姓名，并去掉尾部的空格，然后读取顾客电话号码和所借影碟数，接着读取所借的影碟号。

saveCustomer()方法将顾客姓名、电话号码、所借影碟数及所借影碟号写入到物理文件中。

trim()方法将名字（顾客名和项目名）末尾的空格去掉，因为getline()方法找到输入文件的单词结尾时并不会停下来，它需要给定读入的字符个数或者终止符（回车符）。去掉空格后的字符串作为参数传递。trim()方法不对类的其他数据成员进行处理。这样，

trim()函数就应定义为静态的。而且，trim()函数只在File类的getItem()和getCustomer()方法中被调用，那么应将trim()定义为私有的。

该设计的最高层是Store类，程序14-14是这个类的定义。虽然Store类只给全局函数main()这一个程序组件提供服务，但是为了完整统一起见，还是对条件编译进行一下说明。

程序14-14　Store类的定义（文件store.h）

```
// file store.h

#ifndef STORE_H
#define STORE_H
#include "inventory.h"
#include "file.h"

class Store {
public:
  void loadData(Inventory &inv);
  int findCustomer(Inventory& inv);
  void processItem(Inventory& inv);
  void saveData(Inventory &inv);
} ;
#endif
```

如果不包含“inventory.h”头文件，该文件将无法编译，因为编译程序不能识别Inventory名的意义。但是如果不包含“file.h”头文件，该文件也能编译，因为只在Store类成员函数的实现中才用到了File类（见程序14-15）。

程序14-15　Store类的实现（文件store.cpp）

```
// file store.cpp

#include <iostream>
using namespace std;
#include "store.h"                                    // this is a necessity

void Store::loadData(Inventory &inv)
{  File itemsIn("Item.dat",ios::in);                  // item database
   char ttl[27], category; int id, qty, type;         // item data
   cout << "Loading database ... " << endl;
   while (itemsIn.getItem(ttl,id,qty,category)==1)    // read in
   {  switch (category) {                             // set category for the subtype
       case 'f':  type = 1;  break;
       case 'c':  type = 2;  break;
       case 'h':  type = 3;  break; }
      inv.appendItem(ttl,id,qty,type); }
  File custIn("Cust.dat",ios::in);                    // customer database
  char name[25], phone[15]; int movies[10], count;
  while (custIn.getCustomer(name,phone,count,movies)==1)
  { inv.appendCust(name,phone,count,movies); } }      // pump data

int Store::findCustomer(Inventory& inv)
{ char buffer[200]; char name[25], phone[13];
  int count, movies[10];
  cout << "Enter customer phone (or press Return to quit) ";
  cin.getline(buffer,15);
  if (strcmp(buffer,"")==0) return 0;                 // quit if no data entered
```

```
  bool found = false;
  while (inv.getCustomer(name,phone,count,movies) != 0)
   { if (strcmp(buffer,phone) == 0)                    // search for the phone
      { found = true;  break; } }                      // stop if phone found
  if (!found)
  { cout << "\nCustomer is not found" << endl;
    return 1; }                                       // give up if not found
  cout.setf(ios::left,ios::adjustfield);
  cout.width(22);  cout << name << phone << endl;     // print data
  for (int j = 0; j < count;  j++)
    { inv.printRental(movies[j]);}                    // print movie Id's
  cout << endl;
  return 2; }                                         // success code

void Store::processItem(Inventory& inv)
 { int cmd, result, id;
   cout << " Enter movie id: ";
   cin >> id;                                         // search attribute
   cout << " Enter 1 to check out, 2 to check in: ";
   cin >> cmd;
   if (cmd == 1)
   { result = inv.checkOut(id);                          // analyze return value
     if (result == 0)                                    // not found
       cout << "Movie is not found " << endl;
     else if (result == 1)                               // out of stock
       cout << "Movie is out of stock" << endl;
     else                                                // it is a success
       cout << " Renting is confirmed\n"; }
    else if (cmd == 2)
       inv.checkIn(id);                                  // feedback in checkIn()
    cin.get(); }                                         // eliminate CR from line

void Store::saveData(Inventory &inv)
{ File itemsOut("Item.out",ios::out);  Item item;        // item file
  while (inv.getItem(item))                              // no internal structure
    itemsOut.saveItem(item);                             // save each item
  File custOut("Cust.out",ios::out);                     // customer output file
  char name[25], phone[13];  int count, movies[10];
  cout << "Saving database ... " << endl;
  while(inv.getCustomer(name,phone,count,movies))        // pump data
     custOut.saveCustomer(name,phone,count,movies); }
```

因此，我们完全可以只在文件的实现中包含“file.h”头文件，而不用在Store类的头文件包含它。编译程序可以理解程序。但是从理解的角度来说，这种做法可能并不太好。在一个类的头文件中一次性包含所有使用的服务器类的头文件会更加好一些，因为这样会让维护人员立刻明白这个类使用了哪些服务器类。

但是一些设计人员走了极端，包含了服务器的所有服务器类的头文件，例如“item.h”和“customer.h”。这可能就过头了，它给客户头文件造成了没必要的混乱。

如程序14-14所示，Store类没有数据成员。这对于类层次中的类而言，会出现警告信息，但对于最高层的客户类来说是允许的。Store类的方法负责高级操作，即描述系统的外部界面：在系统开始执行时装载数据库，在数据库中查找顾客，处理顾客借还影碟操作，以及在程序终止时保存数据库。

程序14-15是Store类的实现。loadData()方法创建了一个局部File对象，并将

getItem()消息发送给该File对象，从外部文件读取数据。项目中的每一项数据作为appendItem()调用的参数。将appendItem()消息发送给Inventory对象。loadData()方法将该Inventory对象作为参数，然后创建另一局部File对象，并从文件中读取顾客数据，保存到该Inventory对象中。loadData()方法终止时，这个局部的File对象消失。这将终止物理文件“Item.dat”、“Cust.dat”和File对象之间的联系。

findCustomer()方法提示操作者输入顾客电话号码，如果操作者只按了回车键而未输入任何数据，该方法将终止（返回0）。如果用户输入了电话号码，findCustomer()方法向Inventory对象发送getCustomer()消息，以获取每个顾客数据，这些数据作为参数传给findCustomer()。如果没有找到输入的电话号码，则打印一条错误消息，findCustomer()返回1通知它的客户；否则，打印顾客姓名、电话号码及有关影碟信息，方法返回2。

processItem()方法也有一个Inventory类型的参数，该方法提示用户输入影碟号和借/还命令，然后发送checkOut()消息或checkIn()消息给其参数。如果checkOut()返回，processItem()方法将分析其返回值，并打印相应的消息；如果checkIn()返回，processItem()方法终止，因为checkIn()已对结果进行了分析，并打印了消息给用户。

saveData()方法是loadData()方法行为的镜像，它创建File局部对象，并将saveItem()和saveCustomer()消息及相关信息发送给这个File对象。这些信息是saveData()方法通过使用getItem()消息和getCustomer() 消息从 Inventory参数中抽取而来的。

程序的最后一个组件是Store类的客户——main()函数。程序14-16表明，main()函数实例化了两个对象：一个是 Inventory类的对象，另一个是Store类的对象。该函数将Inventory对象作为参数传递消息给Store对象。

程序14-16　main()函数的实现（文件video.cpp）

```
// file video.cpp

#include <iostream>
using namespace std;
#include "store.h"                                  // this is a necessity

int main()
{
   Inventory inv;  Store store;                     // define objects
   store.loadData(inv);                             // load data
   while(true)
   { int result = store.findCustomer(inv);          // check results
     if (result == 0) break;                        // terminate program
       if (result == 2)                             // 1 if not found
         store.processItem(inv); }                  // process the cassette
    store.saveData(inv);                            // save database
    return 0;
   }
```

图14-12是程序执行的一个示例。这只是一个示例，完整的测试结果太长了。与该实例相应的输入文件如图14-8和图14-9所示，程序执行后所产生的输出文件如图14-10和图14-11所示。

```
Loading database ...

Enter customer phone (or press Return to quit) 353-2566
Shtern                353-2566
101  Splash                          comedy
102  Birds                           horror

  Enter movie id: 101
  Enter 1 to check out, 2 to check in: 2
  Movie is returned
Enter customer phone (or press Return to quit) 353-2566
Shtern                353-2566
102  Birds                           horror

  Enter movie id: 103
  Enter 1 to check out, 2 to check in: 1
  Renting is confirmed
Enter customer phone (or press Return to quit)
Saving database ...
```

图14-12 程序14-6~程序14-16的执行示例

我们假设应用程序实现的类列表与系统要处理的真实生活的实体列表是对应的。正因为如此，才可以从功能说明书的分析中得到这些类。通常，应用程序中类之间的任务划分也非常自然。因此，Item类中所包含的是有关影碟的信息，而不是顾客信息或磁盘文件。

这很自然且比较简单。而位于类层次中最高层的客户类就没那么自然了。如Store类就没有任何直觉上清晰的任务，Store类和main()函数之间的任务划分完全是随意而定的。有人认为main()函数应该没有任务，应该只是对应用程序的顶层对象实例化，应该由构造函数调用激活对象的行为。

如果采用这样的做法，move()方法中的内容要移到Store类的构造函数中。在构造函数中不再需要Store对象，因为在构造函数中，不需要目标对象，Store类的成员函数立即有效。由于只在Store类的构造函数中调用其成员函数，因此，这些成员函数都不必是公共成员，它们可以定义为私有函数。

```
class Store {
private:
  void loadData(Inventory &inv);
  int findCustomer(Inventory& inv);
  void processItem(Inventory& inv);
  void saveData(Inventory &inv);
public:
  Store(void)
   { Inventory inv;                          // define objects
     loadData(inv);                          // load data
     while(true)
     { int result = findCustomer(inv);       // check results
       if (result == 0) break;               // terminate program
       if (result == 2)                      // 1 if not found
         processItem(inv); }                 // process the cassette
     saveData(inv); }                        // save database
   } ;
```

main()函数则变得很简单：

```
int main()
{ Store store;
  return 0; }
```

从美学上而言，这种解决方案更优雅；但从实践上来看，这两种方法不分上下。正如我们前面提到的，类层次中最高层的类与main()函数之间的任务划分是任意的，无法根据系统功能说明的分析进行设计。

14.4　类的可见性和任务划分

程序14-6~程序14-16所描述的实例，提供了讨论类之间关系的很好机会。

本书第一部分中讨论的一个重要问题是函数之间的任务划分，以避免函数之间过多的通信（以及开发人员之间过多的协调和合作）。造成函数之间过多通信的主要原因是：将本该在一起的任务分开，以及将任务上推给客户函数而不是推向服务器函数。

在本书的这个部分，在类之间的任务划分上体现了同样的思想，以避免类之间的过多通信及负责设计这些类的开发人员之间过多的通信。

类之间过多的通信的主要原因是：将本属于一个类的任务拆分在不同的类的不同函数中实现，而不得不通过函数参数和类的数据成员来进行通信。类之间的通信越多，类的设计人员需要考虑的细节就越多，错误的可能性就越大。

在处理类时，也要将客户类的任务尽量交给服务器类。如果不这样做，服务器类会很简单，而客户类却非常复杂难懂。这将给客户端代码程序员和维护人员带来很大的不便，同时也容易出错。

另外一个与类相关而与函数无关的概念是类的可见性（visibility）。客户类所使用的服务器类越多，客户类的设计者和维护人员所要关注的范围就越大，他们不得不研究这些服务器类的界面及使用限制。减少客户类可见的（也就是客户类的设计者应了解的）服务器类数目，可使程序易于理解和维护。

反过来，使用同一个服务器类的客户类越多，程序设计对服务器类的修改就越敏感。减少服务器类可见的客户类数目，可提高程序的稳定性。

当然这些建议在采纳时也不能走极端。毕竟任何程序都可以只用一个类（或者根本没有类）来设计，这样类之间的通信、类之间的任务划分和类的可见性等问题就都消失了。我们想构造拥有互相合作的类的程序，但是我们希望将类之间的通信降到最小。

使用UML类图（类似于图14-4所示）是分析程序结构的一种好方法。类图中类之间的关系说明了类的可见性，它们显示了哪些客户类与某一服务器类相关。遗憾的是，UML图中类之间的关系不能表示类之间的任务划分、将任务推给服务器类以及将本该在一起的任务拆分开等问题。因此，我们必须分析类之间的数据成员和成员函数的分配。与图14-3中内容相似的类图有助于实现这个目的。

14.4.1　类的可见性及类之间的关系

程序14-6~程序14-16所描述的实例中类之间的关系如图14-13所示。

该UML类图说明，Inventory类“拥有”任意多个Item类的对象和Customer类的对象。在设计Inventory类时，我们限制了数组的大小，但这是人为的，与Inventory类和

它所拥有的对象之间的概念关系没有必然的联系。从概念上而言，Inventory类可包含任意多个Item类的对象和Customer类的对象，如图14-13所示。

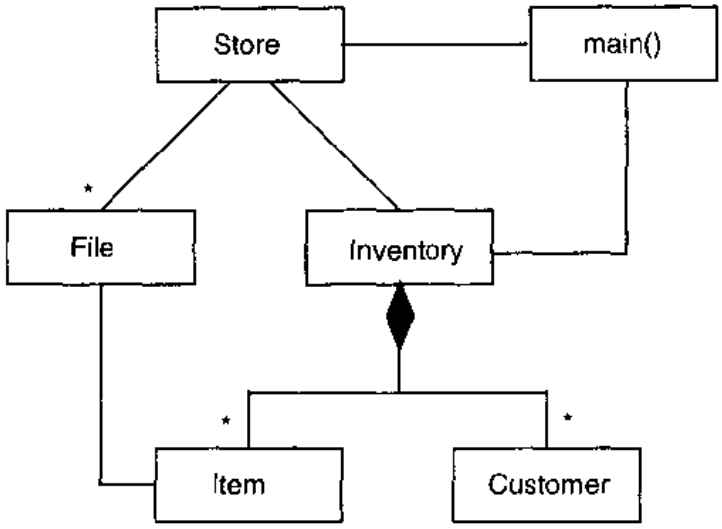


图14-13 程序14-6~程序14-16的UML类图

该类图还说明了Store类是Inventory类和File类的客户，main()函数是Store类和Inventory类的客户，File类是Item类的客户，但不是Customer类的客户。

这是我们在设计Item类和Customer类时，使两者行为不一致导致的结果。Item类的对象知道如何打印自己的有关信息，而Customer类的对象不知道如何处理。因此Customer类提供了getCustomer()方法，客户代码使用该方法获取要打印的Customer类的数据成员。

Inventory类的设计也支持这种不一致。Inventory类的getItem()方法为客户代码提供了Item对象，让客户代码访问Item对象的成员。而Inventory类的getCustomer()方法只为客户代码提供了Customer成员而没有提供Customer对象。因此File类可以“看到”Item类但“看不到”Customer类。

同一程序中，一个类对另一个类的可见性是一个重要的特点，设计者可利用这个特点使类之间的依赖最小，使开发人员之间的协作最少。

将一个对象定义为某客户方法的局部对象时，只有在该客户方法中该对象才可见。这样的协作是很小的。例如，File类的对象只定义在Store类的loadData()方法和saveData()方法中，对于其他类和Store类的其他方法是不可见的。

将一个对象定义为某客户类的数据成员时，该对象对于该客户类的所有方法都是可见的。这种依赖程度要高一些，因为客户方法必须协调使用服务器对象。例如，把Item类和Customer类的对象定义为Inventory类的数据成员，custIdx和itemIdx下标分别指向这些对象。Inventory类的所有方法可以访问这两个数组和这两个下标。这带来了很大的方便性和灵活性，但要求人们之间进行更多的协调。

例如，程序14-11实现了Inventory类，Store类的findCustomer()方法调用getCustomer()方法，将custIdx下标指向Customer对象，该Customer对象参与借还操作。checkOut()方法和checkIn()方法使用同一下标变量custIdx访问同一对象，

但必须减1才能获得该正确的对象。这是一个耦合度的例子，是由于不同的方法访问同一对象所造成的。

如果一个客户对象在它自己的客户方法中定义，该客户对象的服务器类将作为参数传递给该客户对象的方法——这听起来很复杂。例如，图14-13中，Store类是Inventory类的客户，这个客户对象（Store）定义为其客户（main()函数）的局部变量，那么其服务器对象（Inventory类）将作为参数传递给Store的方法。

程序14-16实现了这种关系，main()函数是Inventory类和Store类的客户。它定义了Inventory类和Store类的对象，并将Invertory对象作为参数传递给Store对象。main()函数的设计者和Store类的设计者都要知道Inventory类。

这就是我们所熟悉的信息隐藏，在这里我们称之为对象的可见性。如果Inventory对象被定义为Store类的数据成员，而不是main()函数中的变量，那么只有Store类的方法才能访问Inventory对象。

```
class Store {
  Inventory inv;
public:
  void loadData();
  int findCustomer();
  void processItem();
  void saveData();
} ;
```

这样，main()的设计者就不用知道Inventory类，因此所要关注的范围减小了，编程和维护任务的复杂性也降低了。

14.4.2 将任务推向服务器类

将任务推向服务器类是简化客户代码的好方法，可以消除使客户代码难读的低层细节处理，更好地掌握处理的意义。

例如，程序14-6中Item类提供了getId()和getQuant()方法。这两个方法是通用的，提供了实际的项目号和项目数量。由于其通用性，该设计几乎可以满足任何使用这个数据的需求。

这对类库来说是很好的，我们希望将类库卖给尽可能多的用户。但这样做并不一定适合做程序的一部分，如果我们希望为满足特定客户类的特定需求进行设计，而此客户类属于同一程序或者会在同一程序的下一版本中使用。因为其通用的“库-类型”设计，客户类不得不设计得足够灵活以使用服务器类所提供的服务。通常，服务器类提供的信息要比客户类所需要的多，客户类必须能让这些多余的信息适应当前客户对信息的需要。

例如，在程序14-11中，客户函数printRental()扫描Inventory类中的每个Item对象，并获取到Item对象号（影碟号）的值。现在，printRental()可以对这个值进行任意处理，但它所需要做的只是将这个值与参数值进行比较。

```
void Inventory::printRental(int id)          // used in findCustomer()
{ for (itemIdx = 0;  itemIdx < itemCount;  itemIdx++)
  { if (itemList[itemIdx].getId() == id)
    { itemList[itemIdx].printItem(); break; } }
  itemIdx = 0;}
```

这个信息是冗余的，因为客户代码只想知道下一个Item对象号是否与参数值相同。客户代码获得了比它所需要的（项目号）更多的信息，而它又必须努力处理这些多余信息。一种比较好的任务划分形式是要求客户代码将参数值下传给服务器的函数，让服务器程序代替客户工作（比较项目号），而不是把信息提升到客户代码来处理。这样客户代码变为以下形式：

```
void Inventory::printRental(int id)          // used in findCustomer()
{ for (itemIdx = 0;  itemIdx < itemCount;  itemIdx++)
  { if (itemList[itemIdx].sameId(id))        // important difference
    { itemList[itemIdx].printItem(); break; } }
  itemIdx = 0;}
```

类似地，程序14-10中的客户函数checkOut()调用服务器函数getQuant()来决定当前的项目是否可租。现在，该客户函数可对getQuant()的返回值进行任意处理，但它只是将这个值与0进行了比较。

```
int Inventory::checkOut(int id)              // used in processItem()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
    if (itemList[itemIdx].getId() == id) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx = 0; return 0; }
  if (itemList[itemIdx].getQuant()==0)       // what is the meaning?
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);
  custList[custIdx - 1].addMovie(id);
  itemIdx = custIdx = 0;
  return 2; }
```

同样，这个信息也是冗余的，因为客户代码只想知道这个项目是否有效，但客户代码获得了比它所需要的（当前数量的值）更多的信息，而它也必须处理这些多余信息。一种比较好的任务划分形式是要求服务器的函数去处理与0的比较工作，这样客户代码甚至都不需要知道项目可用性的商业规则。为了避免将信息从服务器提到客户代码处理，服务器类可以提供inStock()函数。修改后的客户代码如下所示：

```
int Inventory::checkOut(int id)              // used in processItem()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
    if (itemList[itemIdx].sameId(id)) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx = 0; return 0; }
  if (itemList[itemIdx].inStock())           // meaning is self-evident
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);             // job is pushed to server
  custList[custIdx-1].addMovie(id);          // job is pushed to server
  itemIdx = custIdx = 0;
  return 2; }
```

注意，checkOut()函数可以保存当前数量值，并检查这个值是否为正数，如果是则将其减1，然后将新的当前数量值保存到Item对象中。这本来是另外一个将任务提升到客户代码完成的例子。但是对于Item对象而言，checkOut()函数并不知道有多少个项目，也不关心确切的数字，而只要知道有项目可以出租就行了。所以，不管数量是多少，都将Item对象的当前数量值减1，然后继续执行。这样，就完成了一个将任务从客户类推到服务器类的例子。

14.4.3 使用继承

在图14-13中的UML图中没有使用继承，因为继承是一种实现技术而不是一种表示真实生活中对象如何互相联系的模型。

使用继承可以简化服务器类的设计，简化客户类的程序代码，减少应用程序中类之间的共享信息。

程序14-6~程序14-16的实例为库存项目实现了一种特有的表现方式。在输入文件与输出文件中，影碟的种类由一个字母表示，例如“f”表示故事片。在显示项目时，影碟的种类由一个单词表示，如“feature”表示故事片。在执行时的内存中，影碟的种类由一个整数表示，如1表示故事片。

商业需求是很普通的。重要的是让服务器类的客户不知道这些行为，但从程序14-6~程序14-16的设计都没很好满足该要求。Item类知道这些行为：其方法printItem()要决定显示哪个单词。Item类的客户File类也知道这些行为：其方法saveItem()要决定将哪个字符写到输出文件中。Store类也知道：其方法loadData()检查要将哪个整数保存到项目内存中供以后使用。只有Inventory类不涉及该问题，但这只是错误地忘记让它涉及该问题而已。

如果设计人员不试图包含类之间的通用信息，这些信息就会在程序中像癌症一样蔓延。客户类的设计人员就无法完成一些与应用程序高层目标相关的建设性工作，而被迫忙于处理一些应该推向服务器类实现的小的细节。

继承是将这些公共信息包含在服务器类中的一个好机制。例如，让Item类作为一些派生类FeatureItem、ComedyItem和HorrorItem的基类，可以将某些特殊项目的行为信息保存在这些派生类中，防止这些信息在程序中到处蔓延。

在本章中没有实现这个解决方案，因为它需要使用多态性，我们会在下一章才讨论多态性。

在从程序14-6~程序14-16的实例中，另一个与使用继承有关的问题是File类的设计。在这个程序中，File类的对象有四种功能：读取项目数据、读取顾客数据、写项目数据、写顾客数据。每个对象都能很好地实现一种功能。例如，程序14-15中saveData()方法的File对象itemsOut只用于写项目数据。如果客户端代码程序员试着要将File消息getCustomer()发送给该File对象，编译程序会接受这个函数调用，但在程序运行时会放弃这个函数调用，因为物理文件只是为写数据而打开的。

注意，如果客户端代码程序员使用该File对象接收saveCustomer()消息，不仅编译程序会接受，而且程序在运行时也不会拒绝。只是将错误的数据写到了输出文件中。

使用继承可以创建特定的类，让每个特定类只负责一种工作。例如，FileOutItem类只将项目数据写到包含项目数据的文件中，而不能读取数据或写顾客数据。

```
class FileOutItem : public File
{
public:
  FileOutItem(const char name[]);
  void saveItem(const Item &item);
   } ;
```

这样，当客户代码试图将getItem()和saveCustomer()消息发送给FileOutItem

对象时，编译程序将提示有语法错误。这样处理太好了。但程序中将包含许多个这样比较小的类，给维护工作增加困难。

一些程序员认为，如果一个File对象是为写项目数据而打开的，那么只有那些能力有限的人和注意力狭窄的人才会试图从文件中读数据，或者写用户数据。理论上这种想法是正确的，这样的错误应该不会发生。但是它们确实发生了。在各种压力下，许多平时聪明和警觉的程序员的注意力也会下降。这样的犯错并没有什么问题。但是，如果否认事实并坚持说有能力的和警觉的程序员就不会犯错，这就是不正确的。更好的做法是面对现实，有意识地避免可能犯错的情况。

图14-14是该实例的UML图。这里，Item类和File类都成了一些特殊类的基类。

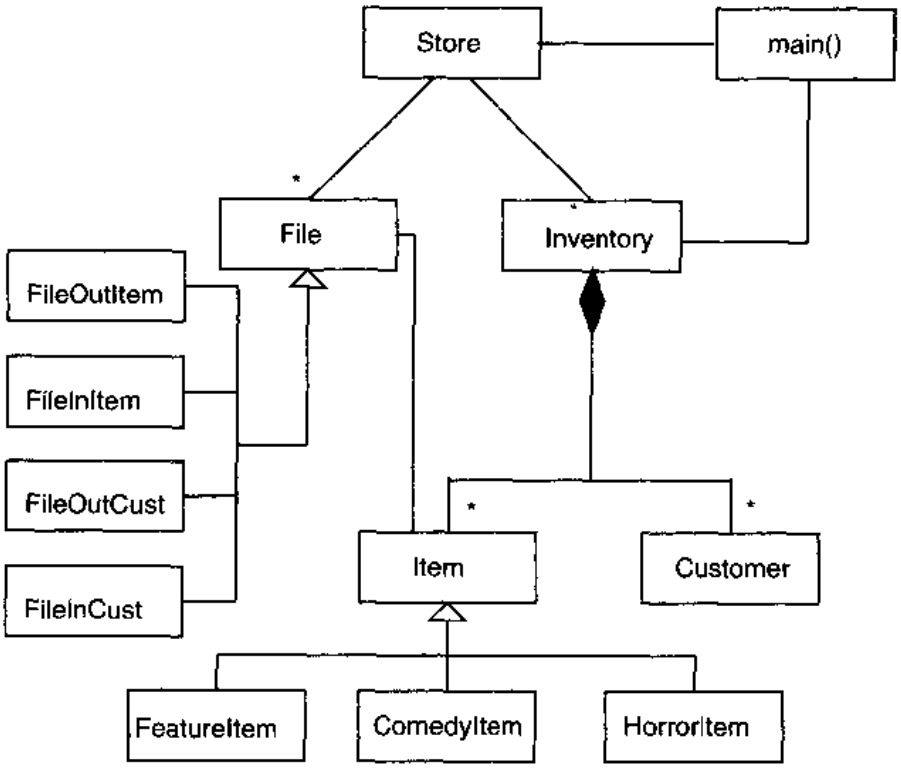


图14-14 程序14-6~程序14-16中使用继承后的UML类图

注意，我们并不提倡这种方案。我们要说明的只是应该考虑使用继承的这种类型。考虑开销时需要权衡的因素有：要实现的类的数目，对防止对象误用的保护度，以及避免在应用程序的类之间扩散公共信息的有效性。

14.5 小结

本章，我们将继承与其他程序设计方法，如类之间的聚集和一般关系等作了比较。

我们强调了其他方法的可行性，因为总的来说继承使用得太多了。的确，使用继承可以简化服务器类设计人员的工作。形式上，客户代码的设计任务也不会因此变得更难，但是这只是表现在代码的编写上没有增加困难，而代码的编写只是程序实现的一个小部分。使用继承迫使客户端代码程序员了解多余的服务器设计的细节，当继承层次很高很复杂时，尤其如此。

我们也讨论了一些使用UML图表示设计的例子，这些UML图有助于设计者看着大的图片讨论类之间的关系。在这些例子中我们只是使用了基本的UML构件，完整的UML是非常复杂的。是否应该马上开始学习UML，还是首先专注于学好C++，这仍无定论。

因为这本书是关于C++而不是关于面向对象设计和分析的，我们更加侧重于C++技能。软件的质量和可维护性取决于程序员编写C++代码的能力，取决于是否能编写出将任务推到服务器类实现的C++代码。至于画复杂的UML图的能力，当然很有用，但是没有掌握C++那么有用。

第四部分　C++的高级应用

本书的最后一部分讨论C++语言的高级应用：虚函数、抽象类、高级重载运算符函数、模板、异常、特殊转换（special cast）以及运行时确认信息。

第15章描述了使用虚函数实现多态机制，这也是一个面向对象程序设计的重要概念。本章首先介绍了相关类与无关类之间的安全类型转换和非安全类型转换的必要（且常与直觉相抵触的）背景知识。然后将它们用于处理不同的（但相关的）类的异构对象列表，该异构对象列表用不同的方法实现相同的操作。随后，介绍了虚函数的语法，使用虚函数将会使客户代码变得非常简单。

此外，第15章还介绍了纯虚函数、抽象类、多继承等概念。虽然虚函数处理异构对象列表很有用，但是夸大了其任务的重要性。多继承也一样，从软件工程角度来看，它的复杂性远远超过了有用性。

第16章讨论了以下内容：一元运算符、下标运算符、函数调用运算符及输入/输出运算符。如果这些运算符与其他重载运算符一起使用，它们可以使客户代码具有良好的语法形式。否则，运算符语法对C++程序质量的帮助是有限的。

第17章介绍了C++重用的其他方法：普通模板。模板定义的语法相当复杂。模板对对象代码的大小及程序运行时间常常有破坏性的影响，初学者应该尽量不要创建自己的模板类。

但C++标准模板库中的模板类设计得非常好，应该尽可能地使用C++的模板类来处理复杂的数据结构。这些模板库类也是设计重用和代码重用的极好例子。

第18章介绍了异常处理，这是一种新的C++方法，也是计算机程序设计一个非常有趣的领域。程序员应该用一定的方式来处理这些异常，从而积累经验，了解这个技术的多用性。另外，这一章还讨论了特殊转换和运行时的对象确认。

第19章是一个总结。这一章的内容一般是一本书的前言中介绍的内容。我们把这些内容放到本书最后是为了确保它不会显得那么空洞。希望大家能够喜欢这个优秀的程序设计语言，并能够有效地使用它。

第15章　虚函数和继承的其他高级应用

在上一章，我们讨论了表示客户-服务器关系的UML符号，研究了在C++程序中实现这些关系的方法。

最普遍的关系是包容（containment）关系（复合或聚集）。实现这种关系的最常用方法是将服务器类的一个对象作为客户类的数据成员。该服务器对象只能由其客户对象使用，而不能和其他客户对象共享。

类之间最一般的关系是关联，如果客户类包含一个指向服务器类对象的指针或引用，那么就实现了类之间的一般关联关系，而服务器对象可被其他客户对象共享。

如果服务器对象只有一个客户，那么这个客户可以专用服务器对象，即使对象之间的关系是一般的关联而不是聚集，仍可将服务器对象作为客户类的数据成员来实现。

将服务器对象作为客户对象的一个数据成员来实现客户-服务器关系，其可见度为中等。服务器对象对于客户类的所有成员函数而言是可见的，但对程序的其他类是不可见的。其他类的设计者不必知道该服务器对象的使用细节，也不需要就其使用情况进行协调。

如果将服务器对象作为客户类的某个成员函数的局部变量实现，其可见度更加有限。这时，服务器对象只对该成员函数是可见的，而对于客户类的其他成员函数和该成员函数之外的其他类是不可见的。如果将服务器对象作为参数传递给客户类的某个成员函数，其可见度要广泛一些。这时，该服务器对象可与客户类之外的许多其他类的对象相关联，这些对象就需要使用该服务器对象进行协调。

将服务器对象定义为服务器方法中的一个局部变量，这种实现关联的方法可使程序组成部分之间的依赖性减少，同时也减少实现和维护的复杂性。将服务器对象作为参数传递给客户方法，这种实现关联的方法提供了更大的灵活性，但对实现人员和维护人员来说可能增加了设计的复杂性。

要选择最合适的方法——即可见度最低但仍能满足客户需要的方法。C++程序员应该考虑从这三种方法中选出一种来实现关联。UML的设计表示符号对这三种方法不加区分。设计人员甚至常常不知道在某种情况下哪种技术最合适。他们仅仅将对象声明为相关而已。因此需要C++程序员进行正确的选择。

我们也讨论了类之间的特殊/泛化关系。使用继承实现类之间的这种关系允许程序员分阶段地创建服务器类，即在基类中实现服务器类的部分功能，再在派生类中实现另一部分功能。这样，继承为C++设计重用提供了强大的、灵活的机制。

本章，将结合虚函数和抽象类来讨论继承的高级应用。在简单应用中，使用继承的目的是使服务器类的设计工作更为简单；而在高级应用中，使用继承的目的是通过简化客户代码使客户代码的设计工作更简单。当客户代码处理的是一组相似对象，而这些对象的操作也相似时，继承的使用显得尤为重要。

“相似对象”指的是它们有一些相同的属性和操作，但在不同类型的对象之间有些属性和操作不同。“相似操作”指客户代码基本上以相同的方式来对待这些不同类型的对象，但根据对象的类型不同，有些操作也会不同。

例如，在前一章的案例分析中，客户代码以同样的方式对待不同种类的库存项目（故事片、喜剧片、恐怖片）：从文件中读取、与租项目的顾客链接、参与借还操作、保存到文件中等。而有些处理根据项目种类的不同而不同，例如，在屏幕上显示项目数据时，根据项目的种类是故事片、喜剧片还是恐怖片而显示不同的标签。

因此，在上一章的客户代码中必须使用switch语句来区别不同种类的库存项目，从而进行不同的特殊种类的处理。这里，我们将使用虚函数和抽象类来简化客户代码，并消除这种对客户源代码的运行时分析。

作为虚函数和抽象类使用的技术基础，我们将首先介绍一个相关问题：在需要某个类对象的地方使用另一个类的对象。C++对这种使用继承进行替换的规则，与对不相关的对象的规则完全不同，也与我们平时对可计算性对象的行为的直觉不同。本章会力图阐明我们应该从哪些方面敏锐地注意这个问题。

在本章最后，我们将介绍当一个派生类有多个基类时，如何使用继承、虚函数及抽象类的技术。

虚函数和抽象类常被认为是C++程序设计的实质，但在实践中并非如此。大多数C++代码处理协作对象而不需要使用虚函数。实际上，大多数C++代码根本就没有（也应该没有）使用继承。尽管如此，用虚函数进行程序设计肯定是有趣的，而且常常是有用的。它是C++中最复杂的论题，希望大家能够学会如何正确地使用虚函数，并从中找到乐趣。

15.1 非相关类之间的转换

正如前面提到的，C++支持强类型概念。这样的现代程序设计原则直觉上很自然而且有道理：如果代码的上下文需要某种特定类型的对象，使用不同类型的对象来代替会出现语法错误。

在如下几种上下文中，这个规则的应用很重要：

- 表达式和赋值。
- 函数参数（包括指针和引用）。
- 作为消息目标的对象。

对于两个不同的类，如果任何一个类都不是另外一个类的直接或间接基类，我们称这两个类为非相关类。注意，这两个不通过继承来互相联系的类也可能有聚集关系或一般关联关系。这也是允许的，但仍不能用其中一个类的对象代替另一个类的对象使用。如果类是通过继承来相关的，就是另外一个问题了。

这里通过一个小例子来演示C++支持强类型的所有三种情况。例子中有两个类：Base类和Other类，这两个类之间不存在继承关系。成员函数Base::set()期望一个整型参数，成员函数Other::setOther()期望一个Base类类型的参数，而Other::getOther()期望一个指向Base对象的指针。为简单起见，我们没有使用引用参数，但所有有关指针的情况也同样适用于引用。

```
class Base {                                    // one class
    int x;
public:
  void set(int a)                               // modifier
  { x = a; }
  int show() const                              // accessor
  { return x; }
} ;

class Other {                                   // another class
  int z;
public:
  void setOther(Base b)                         // modify target
  { z = b.show(); }
  void getOther(Base *b) const                  // modify parameter
  { b->set(z); }
} ;
```

在下面的main()客户函数中，我们定义了三个操作对象：一个为Base类类型、一个为Other类类型，另一个为数值类型。第二行语句是正确但却无关紧要的：其参数的类型是正确的，而且消息目标的对象类型也是正确的。第三行语句也是正确而无关紧要的：表达式的操作数互相兼容，赋值的目标对象也与右值的类型兼容。

这里兼容指的是不同类型的数据值（这里是整数和双精度浮点数）有相同的操作（这里是加、赋值），而且它们之间的类型可以相互转换（这里是整数转换为双精度浮点数，双精度浮点数转换为整数）。

接着的两行语句也是正确的，语句中的消息名（setOther()和getOther()）与目标类（Other）中所描述的成员函数名相同，消息的参数类型也是正确的（第四行是Base类，第五行是Base类指针）。而客户代码中的其他语句都不正确，将它们注释掉。下面我们依次对这些语句进行讨论。

```
int main()
{
  Other a;  Base b;  int x;      // create objects
  b.set(10);                     // OK: correct parameter and target types
  x = 5 + 7.0;                   // OK: right types for expression and lvalue
  a.setOther(b);                 // OK: right type for the target, argument
  a.getOther(&b);                // OK: right type for the target, argument
// b = 5 + 7;                    // not OK: no operator = (int) defined
// x = b + 7;                    // not OK: no operator or conversion to int
// b.set(a);                     // not OK: an object as a numeric argument
// a.setOther(5);                // not OK: cannot convert number to object
// a.getOther(&a);               // no: no conversion from Other* to Base*
// b.getOther(&b);               // not OK: wrong target type, not a member
// x.getOther(&b);               // not OK: a number as a message target
   return 0; }
```

在第一个赋值（如下所示）中，编译程序要求左值为数值类型，而我们用的是程序员定义类型的对象。编译程序要求我们为Base类定义一个带有一个整数类型参数的赋值运算函数operator=(int)，这样该语句就合法了。在第二个赋值语句中，我们将一个程序员定义类类型的对象与一个数值变量相加，这两种类型不兼容。为了让该语句合法，编译程序需要我们为Base类型定义一个operator+(int)函数。在两个语句中，C++的态度都是强制性的：在编译阶段就用强类型排除错误，而不是等到运行时。

```
b = 5 + 7;    x = b + 7;        // syntax errors
```

接下来的两条语句处理参数传递。如果函数的参数是数值类型，例如Base::set(int)，我们就不能使用程序员定义类的对象来代替。转换运算符可能会有帮助，但我们只是在下一章中才讨论。反过来，如果函数的参数是某个特定的程序员定义类型，例如Other::setOther(Base)，则不能使用数值类型或其他的程序员定义类型代替。对于这些情况，编译程序将拒绝进行类型转换，并标识为编译时错误。

```
b.set(a);   a.setOther(5);              // syntax errors
```

C++也努力支持指针和引用的强类型原则。我们把指针和引用放在一起讨论是因为它们的规则是一样的。如果函数的参数定义为指向某类型的对象的指针（或引用），当将指向另一类型对象的指针（或引用）传递给该函数时，会出现语法错误。

```
a.getOther(&a);                         // syntax error
```

显然，当函数的参数为指针时，不能用引用或对象作为实际参数来调用该函数，即使引用或对象与指针属于同一类型。类似地，如果函数的参数为引用，实际参数就不能是指针，即使它们属于同一类型。（但是将对象作为实际参数传递给有引用参数的函数却是可以的。）

对于消息的目标，强类型概念体现为：限制可以被合法地传递给某指定的目标（对象、指针或引用）的消息集合。如果传递给对象的消息名不在对象所属的类定义中，将出现语法错误，无论该函数属于其他类还是不属于任何类。对编译程序而言，它只用知道消息无法在消息目标所属的类中找到就足够了。当然，也不能将消息传递给一个数值变量或一个数据值，因为数值变量或数据值不属于任何类，它们不能响应任何消息。编译程序需要在点选择符（.）左边出现的是程序员定义的类类型的变量。

```
b.setOther(b);  x.setOther(b) ;        // incorrect target types
```

指针（与引用）变量只能指向一个类型的变量，这个变量的类型必须与指针（与引用）定义的类型相同。这也是强类型概念的体现。下面的程序段中，第二行是正确的，第三行是错误的。

```
Other a;  Base b;
Base &r1 = b;  Base *p1 = &b;          // OK: compatible types
Base &r2 = a;  Base *p2 = &a;          // non-compatible types
```

15.1.1 强类型与弱类型

上述规则是理想的状态。但是C++允许这些严格的规则有些例外。一些不受限制的C++性质是从C中继承来的。

例如，从类型检查的角度而言，所有的数值类型是等价的。我们可在表达式中不加区别地自由使用。编译程序将自动地将“较小”的操作数转换为“较大”的操作数，这样所有操作的操作数类型就相同了。对于赋值运算符右边的操作数或函数调用的参数，我们可以在期望是“较小”的数值类型的地方，使用“较大”的数值类型的值。编译程序也会自动地将“较大”的值（如一个长整数）转换为“较小”的值（如一个字符）。编译程序假设程序员知道自己在做什么。

如果在需要一个“较小”类型的数据值时，使用了“较大”的数据值，一些编译程序可能显示一个警告消息。例如，当试图将一个双精度浮点数挤到一个整数中或一个字符变量中时，将会有警告消息。但仅仅是警告而已，不是语法错误。与C一样，C++允许我们使用显式的类型转换，这表示设计人员意图将一个数值类型显式地转换为另一数值类型。

但是采取这种做法的只是那些为维护人员考虑的程序员。对那些以简洁著称的程序员来说，C++再次沿用C的做法，让数值类型之间的隐式转换合法化。这种自由的做法可能会导致赋值或传递参数时的精度丢失。

从这个角度来说，C++（和C类似）是一种弱类型的编程语言。在所有上述情况中，编译程序都假设我们知道自己在做什么，而不对我们的行为做其他猜想。如果我们实际上并不知道自己在做什么，或者没有注意到计算的这些方面，那么我们只能希望实际的计算并不依赖于所截取数据值的精度。

C++也支持强类型规则的其他例外情况，这些情况就不能归咎于向后与C兼容了。产生这些例外情况是因为使用了C++允许我们在类设计中添加的特殊成员函数，也因为使用了如下的强制类型转换：

- 转换构造函数。
- 转换运算符函数。

• 指针（或引用）之间的转换。

这些特殊的函数提供了方法，让编译程序接受违反了强类型规则的客户代码。

15.1.2 转换构造函数

假设Base类提供了一个参数为数值类型的转换构造函数。

```
Base::Base(int xInit = 0)    // conversion constructor
{ x = xInit; }
```

有了这个有效的构造函数，下面的语句将可以编译。

```
a.setOther(5);               // incorrect type, but no syntax error
```

编译程序将以下面的方式解释这条消息：

```
a.setOther(Base(5));         // compiler's point of view
```

这样就创建了Base类的一个临时对象，调用转换构造函数初始化该对象，然后把它作为正确类型的实际参数使用，最后将它撤销。这样，在编译程序级满足了强类型规则：函数得到了它所需类型的值。但在程序员级没有满足强类型规则：程序员将不正确的类型参数传递给了setOther()函数。

注意，我们给这个构造函数提供了缺省的参数值。为什么要这样做呢？在Base类中增加这个构造函数之前，有系统提供的缺省构造函数，我们可以不使用参数来定义Base类的对象。增加了这个转换构造函数后，系统去掉缺省构造函数。在不提供参数的情况下定义Base类对象会出现语法错误。

正如我们在前面提到的，当不从已有的代码中删除任何东西而添加新的代码段（本例中是构造函数）时，C++具有处理异常的能力，即让已有代码语法出现错误。在别的编程语言中，添加新的代码后，可以使程序的无关部分不正确运行，但是不能使它在语法上不正确。从某种观点来看，这是令人沮丧的，因为添加无关的代码不应该导致程序已有的部分出现问题。从另外的观点来看这又是令人兴奋的，因为编译程序可以在编译时而不是运行时就通知程序员出现问题了。

为了避免这些问题，我们也可以为Base类添加一个什么事情都不做的程序员定义的缺省构造函数。这可能是最好的解决方法，因为我们不需要将对象初始化成任何特殊的值。（我们不需要进一步使用这个值）。但是为了省事，我们没有添加另一个构造函数，而是添加缺省参数值0来让已有的客户代码通过编译。这种解决方法的缺点是什么呢？我们造成了假象：会在某处使用这个0值，但实际上不会再用它。我们知道不会再使用它，但是维护人员会猜想使用了它。因此，这样就导致了阅读此代码的难度级别增加（虽然很小）。

所以，将这个转换构造函数增加到Base类后，可用一个数值类型的实际参数调用成员函数Other::setOther(Base)。

```
a.setOther(5);               // the same as a.setOther(Base(5));
```

当编译程序找不到完全相匹配的参数类型时，它将寻找可能的数值类型转换。如果找不到合适的数值类型转换，则寻找数值类型和程序员定义类型的转换的组合，转换构造函数就是一种程序员定义的类型转换。

有了这个转换构造函数，下面的语句也将是合法的，因为编译程序将调用转换构造函数

来满足强类型规则。

```
b = 5 + 7;                    // no error: the same as b = Base(5+7);
```

从某种意义上来说，编译程序是在试图对程序员的做法进行二次猜测。但C++设计的目标之一是避免这样做，而让程序员显式地说明程序代码的意义。其中一种途径是使用显式的转换，以明确地表达我们的意图。但根据C/C++的弱类型规则，数值类型之间的转换不需要显式进行，还可以隐式地调用转换构造函数。怎么办呢？ISO/ANSI标准是一个折中办法。如果类的设计人员认为只能显式地调用转换构造函数，可以使用explicit关键字作为修饰符（详见第10章）。

```
explicit Base::Base(int xInit = 0)          // no implicit calls
{ x = xInit; }
```

explicit关键字的使用是可选的。如果在Base类的设计中使用了这个关键字，将使程序代码更难以编写（使用了这个多余的explicit关键字）。客户代码也会更难编写（客户端代码程序员要使用显式的类型转换），但写出的代码可读性高。如果不使用这个关键字，程序代码的编写会容易些，但代码质量会受到损害。很难找到折中办法。

在C++中，各种转换可以自动地悄悄进行，但explicit关键字将阻止这样做。尽管有了转换构造函数，下面的语句也会出现语法错误，它需要显式地进行转换。

```
a.getOther(5);          // illegal if constructor is defined as explicit
```

注意，隐式转换只适用于按值传递参数，而不适用于按引用或指针传递参数。即使增加了转换构造函数，也不能用数值类型的参数调用Other::getOther(Base* b)。

```
int x = 5;
a.getOther(&x);         // is this is still a syntax error
```

15.1.3 指针或引用之间的转换

隐式转换规则（值的弱类型规则）只适用于数值，而不适用于指针或引用（地址的强类型规则）。但显式转换可适用于任何类型的参数转换。那么能否将一个整型指针传递给需要Base类的指针的地方呢？不能。根据强类型规则，下面这条语句是错误的。

```
a.getOther(&x);             // syntax error
```

但是，可以告诉编译程序这条语句所表示的意义，让它接受这条语句。在C++中办法是显式地转换为正确的类型。

```
a.getOther((Base*)&x);   // no problem, conversion is OK!!
```

在这个函数调用中，创建了一个Base类的指针，并将该指针初始化成指向包含x的内存位置。在getOther()内，把Base类消息传送给x所占的区域。由于Base方法不知道x的数据结构，它们可以轻易地破坏x。整个操作就变得没有任何意义，但在C++中这是合法的。如果程序员坚持这样做，编译程序不会有异议。

任何类型的指针（或引用）转换也类似，不允许不同类型之间的隐式转换。例如，下面的语句是错误的：

```
a.getOther(&a);   // error: no conversion from Other* to Base*
```

getOther()方法所需要的参数是Base类型的指针，而它得到的是指向Other对象的

指针。根据强类型规则，编译程序将此行标注为语法错误：不允许不同类型的指针（或引用）之间进行隐式转换。但编译程序可以接受显式类型转换的函数调用。

```
a.getOther((Base*)&a);  // no problem, explicit conversion is OK
```

这里创建了一个指向Base类对象的指针，并初始化该指针为指向Other对象a。该指针作为实际参数传递给getOther()方法。在getOther()方法中，用该指针来传给属于Base类的Other对象。编译程序不会将这些消息标注为语法错误。但程序执行时可能会崩溃，或无任何警告地产生不正确的结果。这行代码完全没有任何意义，但在C++中是合法的。

15.1.4 转换运算符

转换运算符作为常规的C++转换运算，当应用到程序员定义类型的对象上时，它们通常会返回一个对象组件的值。例如，到int的转换运算符应用到Other对象上时，可能返回数据成员x的值。在需要整数（或其他数值类型）时使用Other类型的对象，运用这个运算符可以消除语法错误。

```
b.set(a);                  // the same as b.set(int(a));
```

当然，并不是我们希望它合法就能合法的。在这个例子中为服务器类Other添加了合适的服务，从而支持了客户代码。第16章中会介绍如何实现这种服务。但是使用转换运算符的思想是清晰的：即对C++系统的强类型转换的另一个对策。

如果希望将Other类型转换为int类型，上面的语句是合法的（使用显式转换更好些）。如果是因为本来是用一个整数而误用了对象a，编译程序不会提示有错误。强类型的保护已不复存在，只有通过运行时测试或调试来发现错误。

总的来说，就数值类型而言，C++是一种弱类型语言。可以自由地将一种数值类型转换为另一种数值类型，而不需要显式地进行。但是如果犯了错误，就要当心后果了。

而就程序员定义的类型而言，C++是一种强类型语言。在数值类型和程序员定义类型之间，或不同的程序员定义类型之间没有提供转换。如果犯了错误，会标注为语法错误，我们可以在运行程序之前进行更正。

转换构造函数和转换运算符削弱了C++系统对程序员定义类型的强类型规则。它们允许在数值类型和程序员定义类型之间进行显式转换，甚至是隐式转换。一旦出错就要小心：它们将不会标注为语法错误。

就指针（与引用）而言，C++提供了一种类似于混合强类型和弱类型的机制。指针（与引用）不能指向与它不同类型的对象。但它们可被任意地转换为其他类型的指针（与引用）。我们要做的只是删除所有显式类型转换（与数值类型值不同，不允许隐式转换，即使是指向数值类型的指针也不允许）。要注意在转换后正确地使用这些指针（或引用）所指向的内存。使用类型转换时要谨慎。

15.2 通过继承相关的类之间的转换

继承使在需要某种类型的对象时使用另外一种类型的对象成为可能。通过公共继承的相关类之间并非完全不兼容，因为派生类拥有基类所有的操作和数据成员。这样可以将一个类的对象赋值给另一个类的对象（可能使用显式类型转换）。当参数是某个类型的对象时，也可

以使用另一个类型的对象（同样地，也可能需要类型转换）。

C++关于通过继承而相关的类之间的转换规则并不复杂。只是看起来似乎违反一般的程序设计直觉。如果真的有这样的感觉，一定要相应地转变这种直觉。本书会努力帮助大家做到这一点。

我们应该知道是什么时候一个派生类公共地继承于它的基类。C++支持从派生类对象到它的公共基类对象的隐式标准转换。从基类对象到派生类对象的转换也是允许的，但是这需要显式类型转换。这个规则适用于类的对象、类对象的引用及对象的指针。

这就是规则。为什么要设计成这样呢？为了让这个正规的规则更加直观，我们会考虑几个例子，并提供一些图形来演示类型之间的转换。

类型转换要阐明的重要概念是安全转换与不安全转换。

15.2.1 安全转换与不安全转换

我们来考虑以下程序段，它使用了数值类型的变量，并演示了不同类型的变量处理。

```
int b = 10;  double d;
d = b;                    // from "smaller" to "larger" type: safe move
```

在这个例子中，我们将“较小”的类型变量（在某台计算机上是4字节）赋给“较大”类型的变量（在某台计算机上是8字节）。无论这个整数变量的值是什么，它将安全地存放到双精度浮点数变量中。在传输过程中，既不会丢失数据也不会破坏数据的准确性。因此，我们认为这种转换是安全的，C++编译程序也不会对这种代码产生任何警告。

下面考虑反方向的类型转换。

```
int b;  double d = 3.14;
b = d;                    // from "larger" to "smaller" type: unsafe move
```

在这里，整数变量中放不下8个字节的双精度浮点数的数据值，其小数部分肯定会丢失。如果双精度浮点数的整数部分超过了整数的合法范围，整数部分也会丢失。因此，我们认为这种转换是不安全的，C++编译程序可能对这种程序代码给出警告消息。

但C++并不认为这种赋值非法。毕竟，不是所有的不安全转换操作都是不正确的操作。double类型的值可能比较小，从而可以容易地存放到整数变量中；或者double类型的值也许没有小数部分，或者小数部分对于应用程序并不重要。

因此，C++把评价情况的任务交给程序员。如果我们知道自己在做什么（即知道转换的值是什么，转换后这个值会发生何种变化），如果我们对结果满意，就没有任何问题。否则，C++不会监督我们的工作。

这些都是C++关于数值类型的变量之间的相关规则背后的逻辑。下面，我们讨论不同类的变量之间的转换。在前一节我们讨论的是无关的类对象之间的转换，接着我们将要讨论的是通过继承相关的类对象之间的转换。

我们来看看Base类（它有一个整数数据成员，大小是4字节）和Derived类（它有两个整数数据成员，大小是8个字节）。

```
class Base {                          // Base class
protected:
   int x;                             // protected data
public:
  Base(int a)                         // to be used in Derived
```

```
  { x = a; }
  void set (int a)                    // to be inherited
  { x = a; }
  int show () const                   // to be inherited
  { return x; }
} ;
class Derived : public Base {
   int y;                                          // in addition to x
public:
   Derived (int a, int b) : Base(a), y(b)          // initialization list
    { }                                            // empty body
  void access (int &a, int &b) const               // additional capability
  { a = Base::x;  b = y; }                         // retrieve object data
} ;
```

我们应用上面“量体裁衣”的规则来在这两个类的变量之间移动数据。

```
Base b(30);  Derived d(20,40);
d = b;                  // from "smaller" type to "larger" type: it fits
```

与前一个使用数值类型的值的例子相似，我们将“较小”类型的变量（4个字节）移到“较大”类型的变量（8个字节）中。在目标对象中有足够的空间存放所赋的值，没有丢失任何数据。

现在反向移动数据：

```
Base b(30);  Derived d(20,40);
b = d;           // from "larger" to "smaller" type: it does not fit
```

这里，我们将较大的Derived类对象的值赋给较小的Base类型的变量。而Base变量没有足够的空间存放Derived值的所有数据成员。这种情况如图15-1所示，它显示了将较小的值赋给较大的值的情况，并标记为安全的。也显示了将较大的值赋给较小的值的情况，并标记为不安全的。

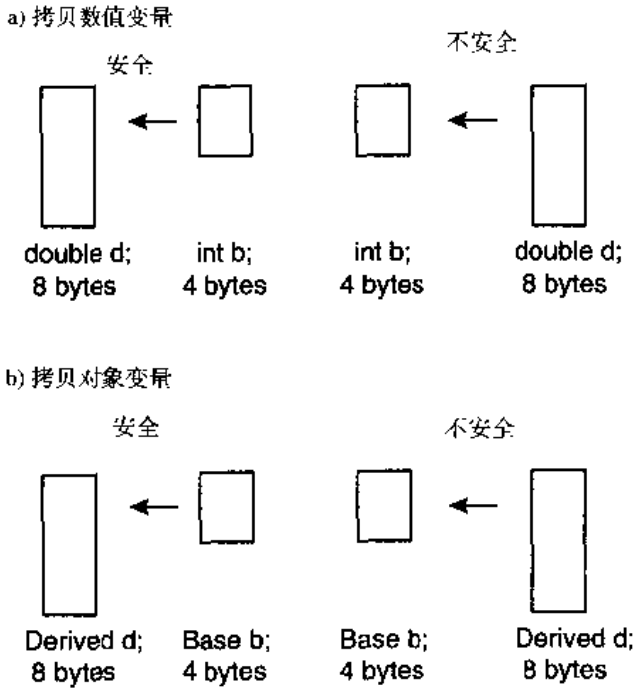


图15-1　不同大小数据值之间的转换：不正确的版本

正如前面提到的，这个逻辑适用于数值类型的变量，但对于有继承关系的相关类的对象不适用。我们需要尽快地调整直觉。类对象的实际问题是对象的一致状态的有效数据，而不是足够的空间。

将派生类对象中的数据赋给基类对象时，派生类对象有足够的数据填充基类对象。它还有多余的数据，但这不是问题。这些多余的数据将会丢掉，因为基类不需要使用它们。这样，基类对象会一直处于一致的状态，因而是安全的。

将基类对象中的数据赋给派生类对象时，基类对象的数据只能赋给派生类对象从基类继承而来的成员，而派生类自己定义的部分则没有赋值，这就有问题。派生类对象最后处于不一致的状态。

因而这就是不安全的，C++声称这种操作有语法错误，图15-2说明了这个问题。对于数据值，问题是需要保持值和精度；对于类，问题是需要有效的数据设置目标对象的所有域。

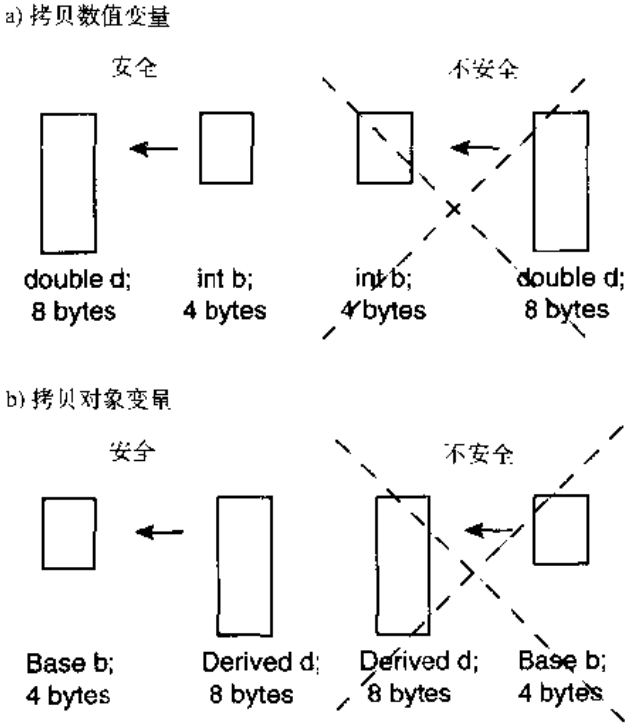


图15-2 不同大小数据值之间的转换：正确的版本

使用显式转换会有帮助吗？毕竟，C++总是提供手段让我们告诉编译程序我们的意图。

```
Base b(30);  Derived d(20,40);
d = (Derived)b;                    // data has nowhere to come from
```

这样仍有语法错误，因为基类对象不能提供丢失的数据。有人可能想将基类对象不能初始化的派生类数据成员（本例中是数据成员y）设置为0。这样做是可以的，但是不能在缺省情况下执行，因为编译程序不知道是否可以接受这些0。为了告诉编译程序，可以为Derived类重载赋值运算符函数：即将此函数的参数定义为Base类，在函数体内复制参数域，然后将剩余的域设置为所需要的值。

```
void Derived::operator = (const Base &b)          // Base parameter
 { Base::x = b.show();  y = 0; }                  // a compromise
```

第11章有一些类赋值运算符的其他例子。在这些例子中，赋值运算符的参数或者与赋值目标的类同一类型，或者与类的一个成员同一类型。这里我们看到的赋值运算符的参数是基类类型。这是可行的，因为重载的赋值运算符也是一个C++函数，而我们可以设计使用所有任何类型参数的C++函数。而且基类对象是派生类对象的一个成员。

现在有两个问题。问题一：为什么这个赋值运算符函数体如此复杂？为什么要这样使用Base类，好像很怕直接使用它似的？为什么不可以用作用域运算符呢？为什么要使用show()函数？毕竟数据成员x在Base类中是受保护的，不是吗？有人可能会说这不止是一个问题，但我们可以将它们归结为一个问题：即能否像下面那样简单地写这个运算符函数？

```
void Derived::operator = (const Base &b)          // Base parameter
  { x = b.x;  y = 0; }                            // nice!!
```

这个问题的答案将在本章后面讨论。

问题二：下面有两行代码试着将Base类的对象拷贝给Derived类的对象，这个赋值运算符函数支持哪一行语句？第一行？第二行？两行都支持还是都不支持？

```
d = b;                  // is this the same as d.operator=(b); ??
d = (Derived)b;         // is this the same as d.operator=(b); ??
```

正确答案是支持第一行。怎么知道的呢？因为第二行没有调用赋值运算符函数，只有第一行才调用了。我们已经强调过一定要记住赋值和初始化的不同，因为在C++中两者是不同的。为了支持第二行，要在编译转换运算符时调用成员函数。这里的转换运算符是什么意思呢？这里的转换运算符指的是调用构造函数。因此，要编写如下形式的代码来支持第二行：

```
Derived::Derived(const ??? &b)      // what type for the parameter?
{ /* what do I do here? */ }
```

我们知道构造函数的名字，那么它的参数是什么类型呢？这种类型应该用来初始化Derived类对象域的类型。根据我们要支持的代码行，这个参数应该是Base类型。

```
Derived::Derived(const Base &b)     // Base parameter
{ /* what do I do here? */ }
```

注意，在这次讨论中用不同的形式多次提及“我们要支持的代码行”，因为是根据要支持的客户代码来决定服务器类是如何编写的。我们应该做的是将参数拷贝给Derived类对象的Base类部分，并为该对象的Derived部分进行一些操作，让该对象处于一致状态。类似于赋值运算符函数，我们可以设置Derived数据成员为0。

```
Derived::Derived(const Base &b)     // Base parameter
{ Base::x = b.x;  y = 0; }          // Hey, this is incorrect!
```

这是不对的。这样做就忽略了要考虑对象构建过程的建议。这样做就误认为构造函数是在对象创建时调用的，而不是在对象创建之后调用。

在创建对象时，先给对象成员分配内存，每个成员的构造函数在给下一个成员分配内存之前调用。对于Derived类的对象，首先为其Base部分的成员分配内存，这时就要调用Base类的构造函数。调用哪一个构造函数呢？由于没有给Base类部分传递任何数据，则应调用缺省的构造函数。让我们检查Base类的定义看看有没有缺省的构造函数。没有缺省的构造函数，只有没有缺省参数值的转换构造函数。这样，试图调用我们正在设计的Derived的构造函数将导致调用一个不存在的Base构造函数，继而出现语法错误。听起来似乎很熟悉？那就对了，我们应该时常考虑这些事情。

这里有什么补救措施呢？我们应该让Base类的缺省构造函数有效，可以在Base类中增加一个缺省构造函数，或者在已有的Base类的转换构造函数中增加缺省参数值。另外一种更好的办法是不在Base类中提供缺省构造函数，而是在我们正在设计的Derived类的构造函数中使用初始化列表。下面我们使用初始化列表：

```
Derived::Derived(const Base &b)                    // Base parameter
  : Base(b.x)                                      // pass data member?
{ y = 0; }                                         // is not this nice?
```

但这样也不正确。Base类的x数据成员不是公共的，因而不能在类作用域之外访问。有人可能认为紧接在Derived::作用域运算符后的构造函数代码是在Derived类中的，而Derived类可以访问Base类的非私有数据成员。这是对的。但Derived类的对象可以访问定义在Base类中的它自己的非公共数据，但参数对象b不同于消息的目标对象，Derived类方法不能访问它的非公共数据。这也是我们在前面所提的第一个问题的部分答案：Derived类赋值运算符函数不能访问其参数Base类对象的内部成员，它只能访问在Base类中定义的它自己的内部成员。因此需要使用Base类的成员函数。

```
Derived::Derived(const Base &b)                    // Base parameter
  : Base(b.show())                                 // pass return value
{ y = 0; }                                         // is not this nice?
```

这是正确的，但是过于细致了。我们调用了Base类成员函数show()，并将其返回值传递给Base类的转换构造函数。调用Base类的拷贝构造函数更简单一些。这个构造函数总是可用的。由于Base类并不动态管理内存，因而没有必要编写自定义的拷贝构造函数。

```
Derived::Derived(const Base &b) : Base(b), y(0)    // copy
{ }                                                // this is really nice
```

至此，我们讨论了将Derived对象复制到Base对象中（没有问题，总是安全的），以及将Base对象复制到Derived对象中（不安全的，除非有拷贝构造函数和赋值运算符的支持，否则这个操作是错误的。）所有这些不仅适用于在赋值运算符中使用对象，也同样适用于将对象按值传递给函数。

在讨论中，我们使用了数据完整性类推办法。我们指出，将Derived类对象拷贝给Base类对象是安全的，因为Derived类拥有Base类的所有数据（还有其他的数据），产生的Base类对象会保持一致的状态。我们也指出，将一个Base类对象拷贝给Derived类对象是不安全的，因为Base类对象是“较小”的对象，它没有初始化较大的Derived对象的必要数据。这样操作会使Derived对象状态不一致。

另一个可能锻炼直觉敏感性的办法是用执行操作的能力来进行类推。在需要一种类型的对象时使用另一种类型的对象可能是安全的，前提是正在使用的对象可以执行所需要的对象可能执行的所有操作。当讨论在需要一种类型的指针（或引用）的地方使用了另外一种类型的指针（或引用）时，我们会演示这个方法。

15.2.2 对象的指针与引用的转换

下一个论题关于使用指向Derived和Base对象的指针和引用。为简单起见，我们只介绍指针的转换情况，但是所有与指针相关的问题也同样适用于引用。我们所讨论的类层次中只

有两个类：Base类和Derived类，但与这两个类相关的所有情况也同样适用于更高、更广的类层次，在那种层次中，基类有其他的派生类，而且派生类又用作其他类的基类。

首先我们动态创建一个Base类的对象，并试图使用Base类指针（这个当然总是可能的）和Derived类指针（这样做可能会有问题）调用其方法。接着，动态创建Derived类对象，并使用Derived类指针（这个当然总是可能的）和Base类指针（这样做也可能会有问题）调用其方法。之后，我们对按指针与引用把参数传递给函数的情况予以总结。

下面是一个在堆中为其分配内存的Base类的对象，我们可以使用指向该对象的Base指针来访问Base的所有方法（如，show()），但不能访问其他类的方法。Derived类的方法（如，access()）不能用Base指针访问，只是因为这些方法不属于Base类。

在处理这些消息时，编译程序首先标识指向未命名的目标对象的指针名（pb），使用指针声明确定指针所属的类（Base），然后进入类定义查找其方法名。如果方法名找到（这里是show()），则生成目标代码，如果没有找到合适标识的方法名（本例中是access()），则出现语法错误。

```
int x, y;  Base *pb = new Base(60);        // base object
cout << " x = " << pb->show() << endl;     // this is OK
pb->access(x,y);                           // this is always impossible
```

接下来，我们设置一个Derived类指针指向同一Base类对象。当该指针（或变量）属于Derived类时，刚才描述的函数名字解析算法被扩展。如果在Dervied类中找到了有消息名的方法则很好；如果没有找到，编译程序将到Base类的描述中查找，如果在Base类描述中仍未找到，则编译程序把此函数调用标注为语法错误。

这样，该Derived类指针可以使用Derived类的任何功能。“任何功能”在这里指的是Derived类定义的功能（如，access()）以及所继承的功能（如，show()）。但事实并非如此。当我们试图将Derived指针指向Base对象时，编译程序立刻就停止编译了。

```
Derived *pd = pb;                          // point to Base object: not OK
cout << " x = " << pd->show() << endl;     // this would be OK
pd->access(x,y);                           // but this should be prevented
```

正式的解释很简单，因为将Base对象转换为Derived对象是不安全的。这样，从一个Base指针（或引用）转换为Dervied指针（或引用）也是不安全的，都会产生语法错误。

但是这个正式的解释并不能让我们更好地理解问题，我们必须培养合适的直觉。不幸的是，用传统程序设计语言进行编程并不会培养与类之间的转换、对象替换、允许和不允许的函数调用等有关的直觉。与对象转换的问题类似，我们也希望能培养关于指针和引用转换的直觉，可以使用以对象大小和为客户代码服务的能力为基础的图形或者类推来达到这个目的。

图15-3中表示Derived指针的矩形比表示Base指针的矩形要大一些，这不是指Derived指针被分配了更多的内存（实际上并没有），而是因为Derived指针比Base指针提供更多的功能。在该分析中，重要的是对象处理消息的能力和为客户代码服务的能力。

Base指针只能访问Base类的功能，Derived指针可以访问Base类的功能和Derived类的功能。我们将动态分配的Derived类对象表示为一个比Base对象大的矩形，不是因为它占有更多的内存空间，而是它具有更多的能力。即使它没有额外的数据成员（在本例中，它有额外的数据成员），但仍有其他的成员函数。在表示Base类对象时，我们使用虚线的部分表示Derived类对象拥有而Base类对象中没有的能力。对象矩形中的B、D字符分别表示对

象中的Base部分和Derived部分。

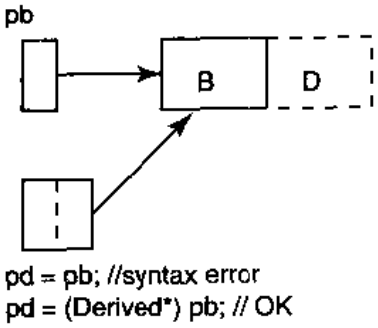


图15-3 通过Derived类的指针访问Base对象：不安全

如图15-3所示，将指针pb的内容拷贝给指针pd，使得这两个指针指向同一Base对象。我们并不是要研究真正的物理地址，或者指针是指向对象的开始还是中间，这些都不重要。讨论的重点是指向对象的指针能根据该指针的类型，而不是按照该对象的类型来激活对象的能力。

我们建议将基类对象看成瘦弱无用的对象，它们处理的事很少。而把派生类对象看成是强大健壮的对象，它们不仅能处理基类对象所处理的事，还能处理其他更多的事。

类似地，我们应该把基类指针看做是瘦弱、作用范围有限的指针，它们看得不够远，只能访问在Base类中定义的方法，但不能访问在Derived类中定义的方法。例如，Base指针可访问在Base类中所定义的方法show()，但不能访问在Derived类定义的方法access()。

最后把Derived类型的指针看做很强大、健壮、有远见的指针，可以看得足够远，可以访问许多方法（Base类中定义的show()与Derived类中定义的access()）。

将强大的Derived类指针指向弱小的Base类对象时，该指针所能提取的信息要比该对象能够提供的多。以上指针赋值语句pd=pb后的两行语句中，第一行（调用show）是对的，第二行（调用access()）不正确，因为在Base中没有access()函数。因此C++认为pd=pb这样的转换是语法错误，以阻止像pd->access(x,y)这样的调用。该调用在语法上是正确的（pd属于Derived类），但语义上没有意义，因为Base对象不能响应该消息。

当然，编译程序也能按照我们的方式理解程序，即Derived指针pd指向一个Base对象，因此调用pd→show()是允许的，但是调用pd->access(x,y)是不允许的。不过，这样的要求对于编译程序的编写者来说太过分了。如果需要在编译程序编写者的利益和C++程序员的利益之间进行选择，C++经常喜欢选择保护编译程序编写者的利益。C++编译程序编写者不需要进行数据流的分析，但是我们需要学习转换的规则。

我们已经知道，pd=pb这样的转换是不安全的，被标注为语法错误。但是如果我们的意图是好的呢？如果我们只是想调用Base的函数（例如show()），而不是Derived的函数（例如access()）呢？那么应有一种机制告诉编译程序一些它不知道但是我们知道的事情，例如我们只使用Base的方法。正如第6章和本章开始处提到的，这种机制实际上已存在，我们称之为转换机制。

使用转换机制，我们要求编译程序执行不安全的转换，因为希望自己控制：调用pd->show()而不是pd->access(x,y)。

```
Derived *pd = (Derived*)pb;       // hey, compiler, I know what I do
cout <<"x="<<pd->show()<<endl;    // I will do this, this is safe
// pd->access(x,y);               // do not even think about this!
```

注意转换运算符名字中不仅包含类名，还要包含指针。如果漏掉指针符号是不正确的。而且不能使用函数符号，只有当类型名是一个标识符时，才允许使用函数符号，而这里的Derived*不是标识符（在C++中，标识符中不能有*符号）。

```
Derived *pd = (Derived)pb;        // cannot convert pointer to object
Derived *pd = Derived*(pb);       // illegal name for functional cast
```

如果要使用函数符号，可以用typedef来定义类型名，例如DerivedPtr。

```
typedef Derived* DerivedPtr;          // new type name: an identifier
Derived *pd = DerivedPtr(pb);         // identifier: OK for this cast
```

下面，在堆中创建一个Derived类对象。使用（强大的、有远见的）Derived类指针指向这个对象，这样就可以激活从Base类继承而来的功能和在Derived类中定义的功能。这是正确的，因为Derived类对象拥有所有这些（强大的）功能。

```
Derived *pd = new Derived(50,80);  // Derived object can do all
cout <<" x="<<pd->show()<<endl;    // OK to call a base method
pd->access(x,y);                   // OK to call a derived method
```

现在，我们将Derived类指针的内容拷贝给（瘦弱的、短视的）Base类指针。这是一个类型转换：Base类指针只可访问它所指向的对象从Base类继承而来的方法。而试图通过这个指针访问在Derived类中定义的方法将是无效的，编译程序对此会产生语法错误。

```
Base *pb = pd;                      // pointer to the same object
cout <<" x="<<pb->show()<<endl;    // sure, Base method is there
// pb->access(x,y);                 // error: not in Base class
```

图15-4显示了代码运行情况。我们再次让表示Derived类指针的矩形比表示Base类指针的矩形大一些，让表示动态分配的Derived类对象的矩形比表示Base类对象的矩形要大。对象矩形内的B、D字符分别表示对象的Base类中的部分和Derived类中定义的部分。

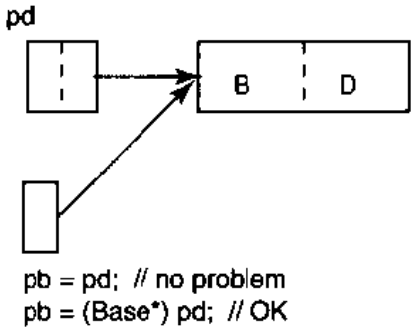


图15-4　通过Base类：安全指针访问一个Derived对象

如图15-4所示，将pd指针的内容拷贝给pb指针，使得这两个指针指向同一Derived类对象（强大的，有力的，可以完成Base对象能够做的所有工作，而且还可以完成更多的其他工作）。但Base类指针pb是瘦弱的、短视的，只能获得对象中的Base类中的功能，而不能获得Derived类中定义的功能。（即使这些功能确实存在、就在该指针指向的对象中。）

将弱小的Base类指针指向强大的Derived类对象时，这个弱小的指针不会造成任何破坏。这个指针只能激活Base类中的方法（如show()）。这些方法总是在强大的Derived类对象之中。因此，C++认为pb = pd这样的转换是安全的，正如它接受将大的Derived类对象拷贝给小的Base类对象一样。这种转换不会导致把一条无法响应的消息发送给未命名的对象。

如果我们想显式地告诉维护人员在编写代码时的考虑（即我们正在将一个Derived指针转换成一个Base指针），也可以使用显式转换。由于这个转换是安全的，可以使用显式转换，也可以不使用。

```
Base *pb = (Base*)pd;                    // explicit cast: alert others
cout <<" x="<<pb->show()<<endl;          // sure, Base method is there
// pb->access(x,y);                      // error: not in Base class
```

这个转换是安全的，但它有不必要的限制。该程序段中调用Derived类的方法access(x,y)被编译程序标注为语法错误。编译程序这样做是因为它知道pb指针属于Base类，而Base类中没有access()方法。由于C++编译程序不进行数据流分析，它有权利不了解我们的认识，即无论pb指针如何弱小和短视，它都指向完整的Derived类对象，这个对象与其他的Derived类对象一样能够响应access()消息。因此，C++编译程序的编写人员会再次遇到困难，必须找到一种办法告诉编译程序这些情况。

我们当然知道如何告诉编译程序这个小指针指向了一个大对象，办法就是使用转换机制。我们必须将这个弱小的Base类指针（它无法获取Derived类的功能）转换为强大的Derived类指针，这样就可以访问Derived类的方法了。

```
Base *pb = (Base*)pd;                    // explicit cast: alert others
cout <<" x="<<pb->show()<<endl;          // sure, Base method is there
(Derived*)pb->access(x,y);               // error: priority of operators
```

这样虽可行，但不是太好。箭头选择运算符→的优先级比转换运算符要高。结果，编译程序会试图将access()方法返回的任何值转换为派生类指针，这会让人迷惑。因此我们要再使用一对圆括号。

这看起来有些难，但是仔细（按步骤）想想，这个语法其实并不复杂。

```
((Derived*)pb)->access(x,y);             // hey, it works!
```

这行代码的确可以运行了，但是看起来很讨厌。因为编译程序无法认识到指针pb指向Derived对象，所以代码才如此复杂。感觉就像在地板上铺上沙，然后再将地板有沙的那一面向下翻转，自找麻烦。

让我们把所有的组成成分放在一起，程序15-1展示了Base类和Derived类，其中客户代码对这两个类的对象进行处理。程序的输出结果如图15-5所示。

程序15-1 使用指针访问基类和派生类的对象

```
#include <iostream>
using namespace std;

class Base {                                          // base class
protected:
   int x;
public:
  Base(int a)                                         // to be used by Derived
```

```
    { x = a; }
    void set (int a)                                    // to be inherited
    { x = a; }
    int show () const                                   // to be inherited
    { return x; } } ;

class Derived : public Base {                           // derived class
    int y;
public:
    Derived (int a, int b) : Base(a), y(b)
    { }                                                 // empty constructor body
    void access (int &a, int &b) const                  // added in derived class
    { a = Base::x;  b = y; } } ;

int main()
{
    int x, y;
    Derived *pd = new Derived(50,80);                   // unnamed derived object
    cout << " 1. Derived pointer, object, and derived method\n";
    pd->access(x,y);                                    // no problem: type match
    cout <<" x = " <<x <<"  y = " <<y <<endl <<endl;    // x=50 y=80
    cout << " 2. Derived pointer, derived object, base method\n";
    cout << " x = " << pd->show() << endl << endl;      // x = 50
    Base *pb = pd;                                      // pointer to same object
    cout << " 3. Base pointer, derived object, base method\n";
    cout << " x = " << pb->show() << endl << endl;      // x = 50
//  pb->access(x,y);                                    // error: no access to derived method
    cout << " 4. Converted pointer, derived object and method\n";
    ((Derived*)pb)->access(x,y);                        // we know it is there
    cout <<" x = " <<x <<"  y = " <<y <<endl <<endl;    // x=50 y=80
    pb = new Base(60);                                  // unnamed base object
    cout << " 5. Base pointer, base object, base method\n";
    cout << " x = " << pb->show() << endl <<endl;       // x = 60
    cout << " 6. Converted pointer, base object, derived method\n";
    ((Derived*)pb)->access(x,y);                        // pass on your own risk
    cout <<" x = " <<x <<"  y = " <<y <<endl <<endl;    // junk!!
    delete pd;  delete pb;                              // necessary tidiness
    return 0;
    }
```

```
1. Derived pointer, object, and derived method
x = 50  y = 80

2. Derived pointer, derived object, base method
x = 50

3. Base pointer, derived object, base method
x = 50

4. Converted pointer, derived object and method
x = 50  y = 80

5. Base pointer, base object, base method
x = 60

6. Converted pointer, base object, derived method
x = 60  y = -33686019
```

图15-5　程序15-1的输出结果

首先，客户代码创建了一个Derived类的对象，用Derived类指针访问派生类方法access()。这是无关紧要的。编译程序在这个指针所指向的类的定义中找到这个方法并调用它，打印x=50，y=80，即第1条输出。

接着，客户代码使用同一个Derived类指针调用Base类的show()方法，编译程序在Derived类描述中找不到这个方法的定义，便查找Base类的定义，找到了该方法后执行调用（并打印x=50）。这里没有用到类型转换。Derived类指针指向的未命名对象属于Derived类类型，它既可以处理来自Base类对象的请求，也可以处理来自Derived类对象的请求（第2条输出）。

随后，客户代码将Base类指针指向Derived类对象。这就不是小事情了，因为这两个指针不属于同一类型。通常情况下，不允许不同类型的指针之间进行隐式转换。由于这两个指针是相关类类型的指针，它们之间的转换是安全的。

```
Base *pb = pd;        // different types: safe for related types
```

这种转换是安全的，因为Derived类指针可以处理Base类指针能处理的任何事情。当客户代码要Base类指针pb处理它不能处理的一些事情时也不会有危险。仍然有一些程序员认为显式的转换比较有用，因为它告诉维护人员（而不是编译程序，因为编译程序知道这些）转换的情况。

```
Base *pb = (Base*) pd;      // related types: cast is optional
```

在程序15-1中，客户代码没有使用这种转换——它是可选的。接着，客户代码使用这个Base类指针调用Base类的方法show()。由于这个Base类指针指向的未命名对象属于Derived类类型，因而将基类消息传送给这个对象没有任何问题。（它打印出x=50，第3条输出）。

然后，客户代码用Base类指针调用access()方法。这里没有安全问题。指针指向Derived类对象也没有问题，它可以处理工作。编译程序看的不是对象而是指针，它在这个指针所属的类（Base类）定义中进行查找，没有找到方法的匹配，因此声明这个调用是语法错误。我们将它注释掉。

客户代码告诉编译程序它所不知道的信息，即这个Base类指针指向一个Derived类对象，但设计人员知道这一点。客户代码通过将Base指针转换为Derived指针来达到这个目的。这个转换是不安全的，要显式说明。转换后的指针属于Derived类，编译程序可使用这个指针调用Derived类方法。

```
((Derived*)pb)->access(x,y);        // we know it is there
```

由于转换后的指针指向的对象实际上是Derived类对象，这个方法调用将正确执行，打印出x=50，y=80，即第4条输出。

接着，客户代码创建一个Base类对象，使用Base类指针调用Base类方法show()。这又是无关紧要的。编译程序在这个指针所属的类定义中查找方法，而不是在这个对象所属的类定义中查找（不用考虑它们实际上是同一个类），找到匹配的方法后调用它。（打印出x=60，第5条输出。）

最后，客户代码做了件糟糕的事情，它忽视了编译程序所允许的自由。它将Base类指针

转换为Derived类指针，这种转换是不安全的。因而需要使用显式转换告诉编译程序它所未知而我们知道且正在处理的信息。我们使用所有合适的括号转换指针，然后调用Derived类方法access()。

```
((Derived*)pb)->access(x,y);          // pass on your own risk
```

在这里，告诉编译程序我们知道正在做什么，指的是我们不会使用这个指针调用Derived类的方法，因为指针所指向的对象只能完成Base类的工作。

既然我们已经告诉编译程序我们知道正在做什么，编译程序不会再做其他猜想，也不会研究Base指针真正指向的对象是哪种类型。这很可惜，因为被Base类对象调用的Derived的方法access()会打印出任何内容（第6条输出）。

这是个很好的例子，告诉我们如何使用完全合法的（但是可能有些复杂的）C++编程模式。

有三个事项值得我们注意。首先，这里与指针有关的所有情况同样适用于C++的引用（惟一区别是一个对象的引用不能再指向另一个对象）。基类的引用不使用转换也能指向派生类对象，只能激活基类定义中的方法，但不能激活基类中没有而在派生类中定义的方法。如果这个基类引用被转换为派生类，它既可以激活基类中的方法，也可以激活派生类中定义的方法。如果不显式转换，派生类的引用不能指向基类对象。使用转换后，派生类的引用可以指向基类对象，并激活所有方法。程序员应该保证不会使用指向基类对象的派生类引用来调用派生。类的方法。

注意　不使用显式转换，基类的指针（与引用）也可以指向派生类的对象。它们只能访问派生类对象中从基类继承而来的部分，因而不会有危害。显式转换是可选的。派生类的指针（与引用）不应该指向基类的对象，因为它们可能会要求这个对象响应派生类的消息。如果觉得不得不将派生类指针（与引用）指向基类对象，则必须使用显式转换。

其次，我们在这里所介绍的有关指针与引用的情况只是适用于相关类。如果两个类之间不存在继承关系，那么不使用显式转换这些类的指针（或引用）不能指向其他类的对象（而且这些转换通常毫无意义）。不允许它们之间的隐式转换，因为这些类没有相同的操作（参见本章第一节）。允许基类指针指向派生类对象的惟一原因是，基类和派生类有相同的操作——即在基类中定义的那些操作。正是这些操作才能被基类指针激活。

警告　某个类的指针（与引用）不能指向与该类没有继承关系的其他类的对象，否则为语法错误。如果不得不这样做，则必须使用显式转换。C++允许这种转换——这很遗憾。将同一对象同时看做两个不同类的对象是自找麻烦。

第三，有继承关系的相关类之间的隐式转换只有当继承是公共继承时才允许。如果是私有继承或受保护继承，所有的转换都必须是显式的。为什么呢？因为在私有或受保护继承模式中，基类的公共操作在派生类中变成私有或受保护的，就不能保证基类和派生类之间有相同的操作。因而，基类指针不能指向派生类对象，基类指针不能访问派生类的操作（C++类的主要属性），派生类对象也不能访问基类的操作（私有和受保护继承的属性）。

因此，只使用公共派生的模式是个好主意。

提示　只使用公共派生来产生派生类。这样允许将一个类的指针（或引用）指向同一

继承层次关系中的另一个类的对象。如果使用私有继承和受保护继承，基类与派生类之间没有相同的公共操作（或数据），将一个类的指针指向同一继承层次关系中的另一个类的对象，就如同将某种类型的指针指向与之无关的类型的对象一样，这是没有意义的，也是自找麻烦

15.2.3 指针与引用参数的转换

现在，我们有足够的知识来讨论函数调用中的参数之间的转换。这里有Other类，它实现了以Base类和Derived类的指针和引用为参数的成员函数。

重载的setB()方法的参数为Base类，将Other数据成员的值设为Base类参数中的数据成员的值。重载的setD()方法的参数为Derived类，将Other数据成员的值设为Derived类参数对象中另外一个数据成员的值。get()方法返回Other对象的内部状态。

```
class Other {                                   // another class
   int z;
public:
   void setB(const Base &b)                     // pass by reference
   { z = b.show(); }
   void setB(const Base *b)                     // pass by pointer
    { z = b->show(); }
   void setD(const Derived &d)                  // pass by reference
   { int a; d.access(a,z); }
   void setD(const Derived *d)                  // pass by pointer
   { int a; d->access(a,z); }
   int get() const                              // selector
   { return z; } } ;
```

这已经足够说明主要问题了。在下面的程序段中，每个函数都获得了在函数接口中所定义的类型的参数，这是最通常的函数使用方法。由于函数期望某个特定的类型的参数，显然在函数内部参数接收的消息也属于相应的类型。在setB()函数中，发送给其参数的是Base类消息show()，在setD()函数中，传递给其参数的是Derived类消息access()。

```
Base b(30);  Derived d(50,80);                  // related objects
  Other a1, a2;                                 // unrelated objects
  a1.setB(b);  a2.setD(d);                      // exact match
  a1.setB(&b);  a2.setD(&d);                    // exact match
```

除了在Derived类中定义的消息之外，参数为Derived类引用的函数还可以把Base类中定义的消息传给这个参数。这不会产生任何问题，因为Derived类的参数可以响应从Base类继承而来的消息（前提是公共继承，而不是私有或者受保护继承）。

对于参数为Base类（指针或引用）的函数，则不能将Derived类的消息发送给这个参数。我们是如何知道这一点的呢？因为Base类的参数不能响应在Derived类中定义的消息，在函数内试图发送这样的消息将不能进行编译。

```
class Other {                           // another class
   int z;
public:
   void setB(const Base &b)             // Base object is expected
   { int a; d.access(a,z); }            // error: Derived message
   . . . } ;                            // the rest of class Other
```

通过对这两种情况的考察，我们可以确定如果不同类类型的指针或者引用作为一个函数的实际参数传递时，会发生什么事情。如果函数的形式参数与实际参数没有继承关系，答案很简单，即如果不使用显式转换，这个函数调用就是一个语法错误，无论参数是按引用还是按指针传递。

```
Account acc1(100), acc2(1000);              // unrelated objects
Other a1, a2;                               // unrelated objects
a1.setB(acc1);  a2.setD(acc2);              // syntax error
a1.setB(&acc1);  a2.setD(&acc2);            // syntax error
```

当按值传递参数时，如果实际参数的类有合适的构造函数，则可用隐式转换。但这不适用于引用与指针参数。消除语法错误的惟一办法是使用显式转换。

```
a1.setB((Base&)acc1);                  // no syntax error but useless
a1.setB((Base*)&acc1) ;                // no syntax error but useless
```

这些函数调用没有语法错误，编译程序认为我们知道自己在做什么。这太糟糕了，因为我们并不知道自己在做什么。在这些函数中，参数Account类的对象要响应Base类的消息，在本例中指的是show()。这在语义上是错误的，程序毫无意义。

当形式参数与实际参数的类型有继承关系时，情况更为复杂。

对于形式参数为Base类指针（或Base类引用）的函数，可以使用Derived类指针（或引用）作为实际参数调用这个函数。这是可行的，因为Derived类对象可以处理Base类对象所能处理的任何事情。形式参数为Base类的函数在函数体中要求其参数只执行Base类的任务。正如我们在前面所提到的，这个函数体内部的Derived类消息将不能被编译程序所接受。

```
void Other::setB(const Base &b)             // pass by reference
{ int a; b.access(a,z); }                   // syntax error
```

上述情况是不可能的，因为C++是一种强类型语言。因此，将Derived类指针（引用）转换为Base类指针（引用）是安全的。在函数内需要Base类对象时，使用Derived类参数对象将不会有问题。

```
a1.setB(&d);                                // safe conversion
```

图15-6演示了这个函数调用。为指针参数b分配空间后，将它初始化为实际参数的内容（粗箭头所示）。这个实际参数是指向Derived类对象d的未命名指针，我们将这个未命名指针标注为&d。这样，两个指针指向同一Derived类对象（细箭头所示）。在函数执行时，把消息发送给参数b，由于这个参数属于Base类，它只能访问对象中从Base类继承而来的消息（对象下的虚线部分）。这个参数指针不能访问对象中的Derived部分的消息，但在函数内没有调用这些消息，因为其参数属于Base类，而不是Derived类。

当函数的参数为Derived类指针（或引用）时，不能用一个Base指针（或引用）作为实际参数调用这个函数。在函数内，由参数指针指向的对象被要求处理的事情只能由强大的Derived类对象完成，不能由弱小的Base类对象完成。

```
a2.setD(&b);                        // syntax error
```

这个转换是不安全的，被标注为语法错误。图15-7演示了这个函数调用。其参数d分配内存空间后，初始化为实际参数的内容，这个实际参数是指向Base类对象b的一个未命名指针。

在函数执行时，把消息发送给其参数d。由于这个参数属于Derived类，它既可以访问对象中Base部分也可以访问对象的Derived部分（虚线表示的）。

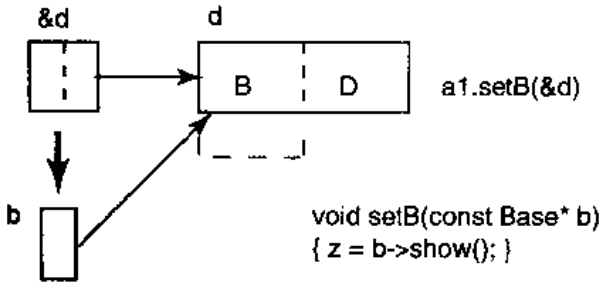


图15-6 在参数传递中将Derived类指针转换为Base类指针

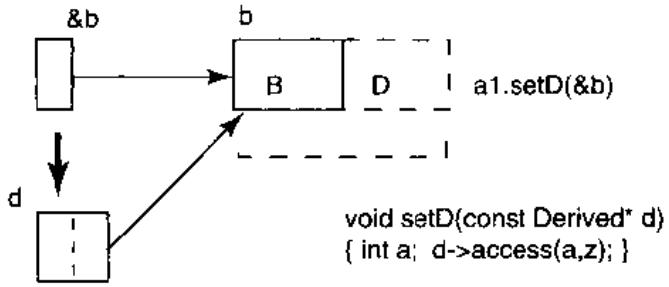


图15-7 在参数传递中将Base类指针转换为Derived类指针

但这个小对象没有任何Derived部分，弱小的Base对象不知如何响应它们。在运行时把消息发送给对象中不存在的部分，其结果是没有定义的。程序可能崩溃或产生错误的结果。

我们假设Derived::setD()方法以不同的方式编写，它只把Base类消息发送给其参数。

```
void setD(const Derived *d)          // pass by pointer
{ z = d->show(); }                   // Base services only
```

将Base类对象发送给这个函数应是安全的，但编译程序不能识别。有些人可能坚持这样做，对此我们可以再次用显式转换告诉编译程序这些信息。

```
a2.setD((Derived*)&b);               // explicit conversion
```

这是告诉编译程序我们知道自己在做什么的通用方法。第一次见到这种转换时，看起来可能让人费解，并有些复杂。但是实际上没有什么神秘的，它只是表示在setD()函数中这个Base对象是安全的。这样小小的Base对象就可以假装成成熟的Derived对象，这完全是可行的，因为在本函数中它只用完成Base类的工作。但是，我们一定要明白自己到底在做什么。

在程序15-2中我们总结了以上讨论的结果。程序的输出结果如图15-8所示。

程序15-2 传递基类和派生类的指针与引用参数

```
#include <iostream>
using namespace std;

class Base {                                    // base class
protected:
   int x;
```

```
public:
   Base(int a)                                    // to be used by Derived
   { x = a; }
   void set (int a)                               // to be inherited
   { x = a; }
   int show () const                              // to be inherited
   { return x; } } ;

class Derived : public Base {                     // derived class
    int y;
public:
   Derived (int a, int b) : Base(a), y(b)
     { }                                          // empty constructor body
   Derived(const Base &b) : Base(b)               // supports implicit cast
   { y = 0; }                                     // explicit initialization
   void access (int &a, int &b) const             // added in derived class
   { a = Base::x;   b = y; } } ;

class Other {                                     // another class
    int z;
public:
    void setB(const Base &b)                      // pass by reference
    { z = b.show(); }
    void setB(const Base *b)                      // pass by pointer
    { z = b->show(); }
    void setD(const Derived &d)                   // pass by reference
    { int a; d.access(a,z); }
    void setD(const Derived *d)                   // pass by pointer
    { int a; d->access(a,z); }
    int get() const                               // accessor
    { return z; }
} ;

int main()
{
   Base b(30);  Derived d(50,80);                 // related objects
   Other a1, a2;                                  // unrelated object
   a1.setB(b);  a2.setD(d);                       // exact match
   cout << " a1=" << a1.get() << "  a2=" << a2.get() << endl;
   a1.setB(d);  a2.setD(b);                                  // implicit conversions
   cout << " a1=" << a1.get() << "  a2=" << a2.get() << endl;
   a1.setB(&b);  a2.setD(&d);                                // exact match
   cout << " a1=" << a1.get() << "  a2=" << a2.get() << endl;
   a1.setB(&d);                                              // implicit conversion
   // a2.setD(&b);                                           // syntax error
   a2.setD((Derived*)&b);                                    // explicit conversion
   cout << " a1=" << a1.get() << "  a2=" << a2.get() << endl;
   return 0;
   }
```

```
a1=30 a2=80
a1=50 a2=0
a1=30 a2=80
a1=50 a2=7011896
```

图15-8　程序15-2的输出结果

在这个例子中，我们使用了与前面例子中一样的Base类和Derived类。其中Derived类多了一个构造函数，我们将马上对它进行讨论。

Other类有两个重载的setB()函数，其形式参数分别为Base类的引用和指针；两个重载的setD()函数，其形式参数分别为Derived类的引用和指针；还有一个get()方法，返回数据成员z的值。

客户代码定义并初始化了一个Base类对象、一个Derived类对象及两个Other对象。第一行输出为a1=30和a2=80，因为在调用setB()和setD()时，函数使用了与形式参数类型完全一致的实际参数。

用Derived类实际参数调用setB()方法不会有问题。不用进行转换Base类引用就可以被Derived类对象初始化，因为这个转换是安全的。用Base类实际参数调用setD()方法导致一个不安全的转换。通常，编译程序会拒绝这种函数调用，认为它是语法错误。使用显式转换后，编译程序就会接受这种函数调用。

```
a2.setD((Derived&)b);        // syntax for casting references
```

当然，这个转换没有任何作用，因为它只是让编译程序接受而已。setD()内的Derived类引用仍然指向小小的Base类对象（如图15-7所示），而且在setD()函数体内的调用Derived:: access()函数访问了并不属于参数对象b的内存。

为了让这个函数调用有意义，setD()中的Derived类参数指针应指向一个Derived类对象，而不是Base类对象。必须用包含在Base类实际参数域中的值初始化Derived类对象，但Derived类对象中有些域是Base类对象中所没有的。应将这些域设置为某些合适的值，如0或其他适合应用程序的值。

那么，Derived类应提供哪个函数来确保正确地初始化对象域呢？那就是构造函数。用哪个构造函数呢？该构造函数名依赖于其参数的个数及类型。由于这个构造函数使用Base类对象的数据初始化Derived类对象的域，它只需要一个参数，该参数的类型是Base类（或Base类引用）。因此，这个构造函数名应是转换构造函数。

这是个恰当的构造函数名，它很适合于将一个Base类的值转换为Derived类的值的构造函数。

```
Derived::Derived(const Base &b) : Base(b)   // copy constructor
{ y = 0; }                                  // explicit initialization
```

这个转换构造函数可以显式调用，创建一个临时Derived类变量，该变量由构造函数初始化，setD()内的引用参数指向它，在调用setD()后将该变量删除。

```
a2.setD((Derived)b);              // explicit constructor call
```

这意味着这种方法只适用于输入参数。如果用于输出参数，需要在函数内对在函数调用后被撤销的临时副本进行修改。

由于我们未将该构造函数定义为explicit，因而可以隐式调用它，正如程序15-2中的客户代码所示。第二行输出为a1=50和a2=0。

客户代码的第二部分处理的是指针参数而不是引用参数。首先用确切的参数匹配调用setB()和setD()，输出结果为a1=30和a2=80。用Derived类指针作为实际参数调用setB()不会有问题，从Derived类转换为Base类是安全的。而用Base类指针作为实际参数调用setD()将会出问题，因为这个转换是不安全的，被标注为语法错误（我们将它注

释掉）。

为了让编译程序接受这个调用，我们使用显式转换将Base类指针转换为Derived类指针。这将方便编译，但不创建Derived类对象。参数为Base类指针的构造函数（类似于引用参数的构造函数）可以解决这个问题。但这些构造函数不常用，因此我们决定让本例产生一些莫名其妙的结果，以再次说明从Base到Derived的转换是不安全的。

注意　将一个派生类指针（或引用）作为参数传递给以基类指针（或引用）为参数的函数，这总是安全的（是否进行转换是可选的）。而将基类指针（或引用）作为参数传递给以派生类指针（或引用）为参数的函数，就会出现语法错误。将基类指针（或引用）转换为派生类指针以迫使编译程序接受这样的函数调用是不安全的。

我们花费了大量的时间，严肃地强调了在不同类类型的对象（或者指针或者引用）之间进行转换的问题。对于不相关的类来说，这个论题没有任何实际的重要性。很少有人在需要某种类型时使用另一种无关的类型。当出现这种极少出现但似乎合法的需求时，我们可以改变一下想法，看看这种需求是否消除了，我们可能会发现不用转换也可以消除问题。在无关的类型之间进行转换是令人困惑和危险的。

对于有继承关系的类而言就完全是另外一回事了。在C++程序设计中，在需要继承层次结构中的某种类型时使用另一类型，这种情况非常普遍。实际上，在任何面向对象的程序设计中都很普遍。因此，理解这种技术及其限制和含意是很重要的。

在有继承关系的类类型之间进行转换也是令人困惑的，也是很危险的。它们不符合数值之间转换的通用程序设计直觉。希望这些讨论能够帮助大家培养一些直觉，知道什么是合适的、什么是不合适的。评估转换的主要标准不是看转换后的结果是否有足够的空间容纳源类型，而是看转换是否是安全的，转换后的结果是否会被要求做一些它不能完成的工作。

将派生类类型转换为基类类型是安全的，将基类类型转换为派生类类型是不安全的。

充分理解了这些以后，下面我们讨论C++的虚函数。

15.3　虚函数

在C++中，每个可计算对象都有定义对象类型的属性。对象由对象名（标识符）及与标识符相关的类型来表示。本书中，在声明或定义对象而指定对象的类型时，标识符与类型是相关的。这对于变量和函数也成立。

在定义或声明变量时，源代码必须指定可计算对象的类型。在程序执行过程中，对象名与类型之间的关系不能被打破。程序可以用同一标识符和同一（或者不同）类型定义其他的可计算对象。这是可以的，但是其他的对象就是其他的对象。即使使用相同的名字也表示不同的可计算对象。

对于函数也是类似的。函数声明（函数原型）或函数定义（函数体）包含一个表示函数名的标识符。这个函数名与该函数产生的目标代码相关联，在程序执行过程中不能破坏或修改这种关联。程序可能用同一函数名定义不同的函数。这些函数可能在同一类中（有不同标识），或者在不同类中，或在不同作用域中（有相同或者不同的标识）。但它们是不同的函数，只是函数名相同而已。

实际上，C++函数名重载并没有违反C中的惟一函数名规则。对于我们而言，在同一类中或不同作用域中重用函数名，函数名是相同的。而对编译程序来说，所有这些函数有不同的

名字，这个名字是函数所属的类名、函数标识符、返回类型及参数类型的串接。

因此，当编译程序遇到一个函数定义时，它将类名、返回类型及参数类型添加到函数标识符中，从而为该函数创建了另一个名字。这样，每个函数名对于编译程序来说是惟一的。这个技术称为名字杂凑技术（name mangling）。

例如，在几个类中都定义了draw()函数，但这些函数有不同的标识（signature）。每个函数表示一个单独的计算实体。

```
class Circle {
    int radius;
public:
    void draw();
    . . . . } ;              // the rest of class Circle

class Square {
  int side;
public:
  void draw();
  . . . . } ;                // the rest of class Square

class Rectangle {
  int side1, side2;
public:
  void draw();
  . . . . } ;                // the rest of class Rectangle
```

我们经常理所当然地认为，函数名与可计算对象之间的关联是在编译时产生的，因为编译程序在编译时处理可计算对象的定义或声明。

例如，下面的客户代码定义了Circle、Square及Rectangle对象，并将它们画在屏幕上。

```
Circle c;   Square s;   Rectangle r;     // name/type are connected
c.draw();   s.draw();   r.draw();        // name/function are coupled
```

编译程序（与维护人员）知道对象c的类型是Circle，对象s的类型为Square，对象r的类型是Rectangle；也知道第一个draw()调用指的是调用Circle::draw()，第二个draw()调用指的是调用Square::draw()，第三个draw()调用指的是调用Rectangle::draw()。

在像C++这样的语言中，我们不会在程序执行过程中改变这些编译时的关联，程序中对象的类型描述了对象的固有属性，这又是强类型的一个表现。

现在表达式及参数传递中的强类型规则是理所当然的。对于每个可计算对象，编译程序和客户代码的设计人员（以及维护人员）事先就已经知道其合法操作集。

强类型提供了所谓的早期绑定(early binding)。可计算对象的类型在编译时就固定下来，程序执行过程中不能改变。它的另一个通用术语是静态绑定（static binding）。两者有相同的意义，即对象名字和对象类型在编译时就固定下来，而且不会在程序执行过程中动态地修改。

当（由消息名及实际参数列表定义的）消息传送给对象时，编译程序根据该对象的类（类型）解释这条消息。这个对象的类名在编译时已确定，在程序执行过程中不能改变。

静态绑定是C++、C、Java、Ada、Pascal（但是没有Lisp）等现代语言的标准之一。它最先引入是为了提高程序的性能，而不是程序的质量。动态绑定，即在运行时查找函数调用的意义，就需要消耗时间。当函数调用在编译时就已确定时，会增加编译时间，但它能提高程

序执行的速度。

以后，我们还会发现静态绑定能成功地用来加强类型检查。如果用错误的参数个数或参数类型调用函数，这个调用在编译时就会被拒绝，而不是等到程序执行时才显示不正确的行为。如果消息名（带有正确的标识）在类定义中没有找到，这个调用也会在编译时拒绝，而不是在运行时。

强类型提供了编译时类型检查，改善了运行时性能，可用于大多数应用程序。

我们还想在其他什么时候建立标识符与可计算对象之间的关联呢？答案是：在编译后，运行时。这样做可能带来什么优势呢？

考虑异构对象列表的处理，或不同类型的对象的外部输入流的处理。通过文件输入或用户交互输入，程序并不知道实际传来的确切对象类型。

例如，程序可以在屏幕上一个接一个地画不同形状的图。根据每个图形的实际特点，程序应调用Circle::draw()、Square::draw()和Rectangle::draw()，或者其他任何已知的图形。但是，如果我们在源代码中只用一条语句，再根据shape对象的特征改变这条语句的意义，这就更好。

```
shape.draw();     // from class Circle, Square, or Rectangle
```

如果当前通过循环传递的对象shape是Circle，则这条语句将调用Circle::draw()。如果是Square，则这条语句将调用Square::draw()。如果是Rectangle，则这条语句将调用Rectangle::draw()，等等。

对于强类型机制，这是不可能的。编译程序将查找变量shape的声明，确定它所属的类，查找类定义。如果在类中没有找到无参数的返回为空的函数draw()，编译程序将产生错误消息。如果找到了，则生成目标代码。但draw()函数的类型要在编译时就确定下来，根本就不能在运行时再查找函数draw()的意义。

我们在这里用到的是运行时绑定，或称为晚期绑定（late binding），或称为动态绑定（dynamic binding）。我们的确假设存在几个可计算对象（不同类中的draw()函数），并希望在某个特定的函数调用中将其中一个可计算对象绑定到draw()。我们想让这个函数调用中的draw()代表Circle:: draw()、Square:: draw()、Rectangle:: draw()等。我们希望到运行时才确定函数的意义，而不是在编译时。这样可以根据函数调用的意义将不同形状画在屏幕上。

对术语再进行一些说明。确定函数名所代表的意义的科学术语是绑定（binding）。编译程序将函数名绑定到某个函数上。我们希望这种绑定在运行时进行，因而被称为运行时绑定而不是编译时绑定。我们希望这种绑定在编译之后发生，因而也称为晚期绑定而不是早期绑定。我们希望这种绑定允许根据使用对象的不同特点，用相同的函数名代表不同的意义，因此称为动态绑定而不是静态绑定。

在函数调用中用同一函数名代表不同的意义，我们将这种能力称为多态性（polymorphism，即“多种形式”的意思）。一些作者在更加广泛的范围意义上使用这个术语，包括在不同的类中使用相同的函数名，这根本没有使用动态绑定。我们不想争论到底哪个定义更加正确（或者更加有用）。我们也不会多次使用这个术语。本书中的多态性指的是晚期绑定或动态绑定，在运行时根据消息目标对象的实际类型确定函数调用的意义。

这正是设置C++虚函数的目的。

15.3.1 动态绑定：传统方法

当然，动态绑定并不只是面向对象程序设计才有的问题。处理异构集合一直是共同的计算任务，程序员们通常在任何可用的语言中实现动态绑定。在各种情况下都要处理相似对象，这些对象非常相似，可以在不同种类的对象中使用同一名字代表某个函数（如draw()），但这些对象的类型是不同的，每个函数以各自不同的方式工作。

下面我们讨论处理一个大学数据库中的实体列表的例子。为简单起见，假设只有两种类型的记录：学生和教师。假设这个程序只维护三种信息：学校代号、名字和职称（对于教师而言）或专业（对于学生而言）。如图15-9的数据小例子。

```
FACULTY
UI2345678
Smith, John
Associate Profesor
STUDENT
UI2345611
Jones, Jan
Computer Science
FACULTY
UI2345689
Black, Jeanne
Assistant Profesor
STUDENT
UI2345622
Green, James
Astronomy
```

图15-9 动态绑定的输入数据

代号值的长度对于每个人都是相同的（9个字符），可实现为固定长度的字符数组。名字、职称和专业对于不同的人有不同的长度，将它们作为动态分配的数组来实现比较合适。下面是每个人的数据结构：

```
struct Person {
       int kind;                    // 1 for faculty, 2 for student
       char id[10];                 // fixed length
       char* name;                  // variable length
       char* rank;                  // for faculty only
       char* major; } ;             // for student only
```

当然，我们可以将这个结构用带有构造函数、析构函数及成员函数的类来实现，但是在讨论的现阶段，这种机制只会让例子更加模糊。我们会在后面讨论更加现代的方法时介绍这些内容。

首先，更传统的方法是，将不同种类对象的特征（如rank、major）合并到一个类定义中。为了以不同的方式处理每个对象，我们增加了一个域来描述特定对象所属的种类。在客户代码中，我们可以使用switch或if语句的分支来实现不同种类对象的处理。

其实还可以使用union结构而不需要为两种对象定义域。这对于大小固定的数组才有意义。但是对于动态内存管理而言，我们认为每个对象一个指针节省下来的内存，并不能抵消使用union带来的额外复杂性。

动态内存管理可节省存储空间，防止内存溢出。将数据保存在内存中的简单方法是定义

一个Person对象的数组。尽管我们使用了struct关键字，Person类型的变量仍可作为对象看待。因为在C++中，struct和class关键字是同义词（除了缺省访问权限和缺省继承外）。

```
Person data [1000];              // array of input data
```

将数据存放在对象的指针数组中，比存放在对象数组中更为灵活。分配一个较大的指针数组的开销并不很昂贵。为了防止溢出，可以为指针数组重新分配内存而且不需要拷贝已有的数据（见第6章中的例子）。每个Person对象在从输入文件读取数据之后，在堆中分配空间。

```
Person* data [1000];             // array of pointers
```

为了从输入文件中读取数据，我们定义了一个库类ifstream的对象，这个对象为了输入而总是处于打开状态。为了让物理文件与这个逻辑文件对象相关联，必须将物理文件名作为调用构造函数的参数。

```
ifstream from("univ.dat");          // input data file
if (!from) { cout << " Cannot open file\n"; return 0; }
```

对于输入文件中的每个对象，程序将动态地分配结构，然后读取四个数据项：定义对象类型的字符串、代号、名字、职称或专业。为了正确存储输入数据，程序将检查定义对象类型的字符串的值（“FACULTY”或“STUDENT”），并将对象的kind域设置为1或2。

```
char buf[80];                       // buffer for input data
Person *p = new Person;             // allocate space for new object
from.getline(buf,80);               // recognize the incoming type
if (strcmp(buf, "FACULTY") == 0)
  p->kind = 1;                      // 1 for faculty
else if (strcmp(buf, "STUDENT") == 0)
  p->kind = 2;                      // 2 for student
else
  p->kind = 0;                      // type not known
```

由于id域的长度已知，可将它直接读到Person对象的域中。而name、rank与major数据的长度在数据读入内存之前是不可知的。因此，程序应将这些数据放到固定大小的缓冲区中，计算其长度，分配足够的堆内存，再将数据从缓冲区中拷贝到堆内存中。

```
from.getline(p->id,10);                    // read id
from.getline(buf,80);                      // read name
p->name = new char[strlen(buf)+1];         // allocate space
strcpy(p->name, buf);                      // copy name
from.getline(buf,80);                      // read rank/major
if (p->kind == 1)
{ p->rank = new char[strlen(buf)+1];       // space for rank
  strcpy(p->rank, buf); }                  // copy rank
else if (p->kind == 2)
{ p->major = new char[strlen(buf)+1];      // space for major
  strcpy(p->major, buf); }                 // copy major
```

这个例子中的内存结构如图15-10所示，左边的data[]是栈数组，该数组右边的其他所有内存（Person类型的对象及其动态内存）都是在堆中分配的。

将读取算法封装在一个函数中，例如read()，这是个好主意，这样客户代码将文件对象和Person指针传递给这个函数。read()函数应为Person对象分配内存，从文件读取数

据，并用输入数据填充Person对象。

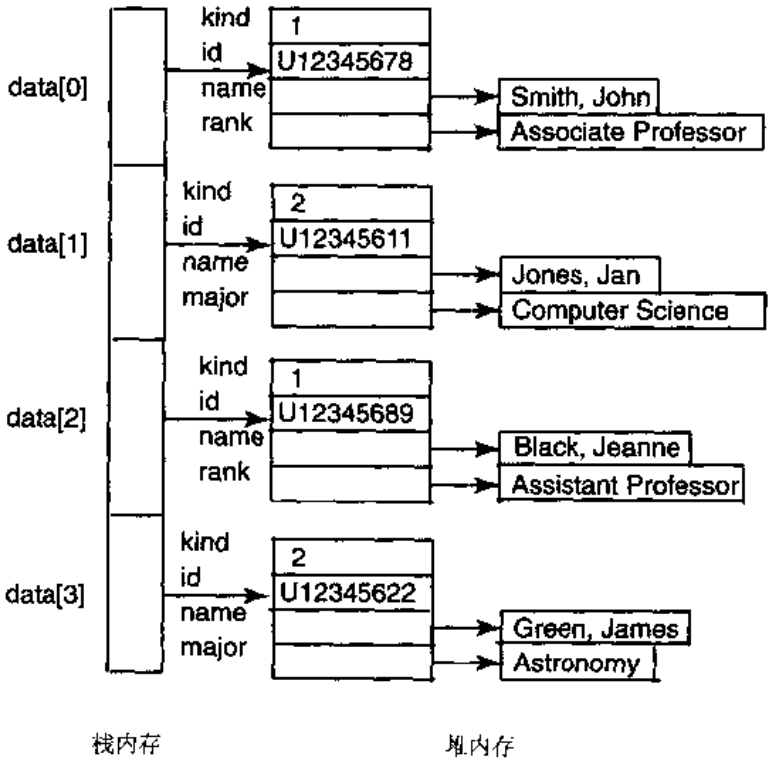


图15-10 为输入数据动态分配的内存结构图

```
Person* data[20]; int cnt = 0;          // array of pointers
ifstream from("univ.dat");              // input file: a library object
if (!from) { cout << " Cannot open file\n"; return 0; }
while (!from.eof())                     // read until eof
{ read(from, data[cnt]);                // data[cnt] is of type
Person*
  cnt++; }
cout << " Total records read: " << cnt << endl << endl;
```

现在，我们把前面所讨论的内容组装成read()函数。这个函数中有两个主要不足，它们都与参数传递有关。

```
void read (ifstream f, Person* person)          // bad interface
{ char buf[80];
  Person* p = new Person;                       // allocate space for new object
  f.getline(buf,80);                            // recognize the incoming type
  if (strcmp(buf, "FACULTY") == 0)
       p->kind = 1;                             // 1 for faculty
  else if (strcmp(buf, "STUDENT") == 0)
       p->kind = 2;                             // 2 for student
  else
       p->kind = 0;                             // type not known
  f.getline(p->id,10);                          // read id
  f.getline(buf,80);                            // read name
  p->name = new char[strlen(buf)+1];            // allocate space
  strcpy(p->name, buf);                         // copy name
  f.getline(buf,80);                            // read rank/major
```

```
  if (p->kind == 1)
{ p->rank = new char[strlen(buf)+1];        // space for rank
  strcpy(p->rank, buf); }                   // copy rank
else if (p->kind == 2)
{ p->major = new char[strlen(buf)+1];       // space for major
  strcpy(p->major, buf); }                  // copy major
person = p; }                               // hook it up to array
```

在这里，我们按值把参数传递给这个函数。这对于文件对象来说是显然的。当从文件中读取数据时，改变了文件对象的内部状态。如果内部状态因为某种原因没有改变，则下次读取的是同一数据，而不是下一个记录。当文件对象按值传递时，形式参数对象的内部状态会发生改变，但实际参数文件对象的内部状态保持不变。正如我们以前说的，不应该按值传递对象，而要按引用传递参数。

```
void read (ifstream& f, Person* person)      // read one record
{ char buf[80];
  Person* p = new Person;                    // allocate space for new object
  . . .                                      // the rest of read()
  person = p; }                              // hook up new object
```

我们坚持认为，如果函数执行过程中不改变对象的状态，则应将对象引用标注为常量。这里没有使用const修饰符，因为从文件读取数据时文件对象发生了改变。作为服务器类ifstream的客户端代码程序员，我们不需要知道其设计细节，只需了解文件对象的状态反映了物理I/O操作的结果就足够了。

下面我们来看一下指针参数。在C++中，如果按指针（或引用）传递参数，这个参数的值在函数内可以改变，而且这个改变还会影响客户代码的实际参数。注意，这里所说的是“按指针传递的参数”，而不是指针参数，这正是一些程序员觉得困惑的地方。上面的read()函数中，Person类型指针person不是按指针来传递，而是按值传递。因此，它的值在函数调用中不能改变。如果在函数调用之前，这个参数指针没有指向某个对象。在函数调用之后，这个指针依然如此，不会指向新分配的Person对象。

那么，这里按指针传递的是什么呢？形式上可以认为指针传递的是一个Person对象（而不是指针）。实际上，如果能适当地将一个Person对象传递给read()函数，它也可用读取的文件数据正确地填充。

```
Person person;                  // Person object, not pointer
read(from, &person);            // object is passed by pointer
```

所谓“适当地传递”指的是传递对象的地址，这个对象存在于客户代码的内存空间。按指针传递对象允许在函数调用过程中改变对象的状态。即使这样解释，read()函数仍有一个问题：person变量现在的类型是Person，但read()中最后一行所用的变量p的类型是Person*。正如我们在第6章中强调的，这两个类型看起来很相似，但是不能忽视两者之间小小的区别。它们是两种不同的类型。其中一个是有所有成员的类，另一个是指向类对象的指针。如果需要可以进行修改，但是我们将要面对客户空间中的这个Person对象数组，而不是一个Person指针数组。

那么，怎么样修改read()函数，从而使该函数中分配的所有堆内存都能正确地链接到如图15-10所示的栈内存中呢？。

在调用read()函数时，Person对象仍不存在，它将在函数内被分配。这个指针指的

是存在于客户空间中的一个Person对象（整个指针数组）。该指针将传递给read()函数。

```
while (!from.eof())                    // read until eof
{ read(from, data[cnt]);               // data[cnt] is of type Person*
  cnt++; }
```

在调用这个函数之前，实际参数Person指针没有包含什么内容，因为它是堆内存的一部分。在调用之后，这个指针指向在read()函数内分配的Person对象，如图15-10所示。这样，它包含了一个有效的堆内存地址，其内容在调用后被改变。然后这个指针应按引用或按指针传递。因此上面的read()函数是错误的。

要注意分清以下术语：指向对象的指针、指针的引用、指向指针的指针等。记住，指针是一个普通的变量，因此它既可以按值传递，也可以按引用或指针传递。只是在C++中，这个函数接口有两种不同的解释。

```
void read (ifstream& f, Person* person)      // read one record
  { char buf[80];
    Person* p = new Person;                  // allocate space for new object
    . . .                                    // the rest of read()
    person = p; }                            // hook up new object
```

在这里，Person* person既可解释为按指针传递一个Person对象，也可解释为按值传递Person指针（其类型为Person*）。按引用传递这个指针也根本不困难。标准的C++规则（如第7章中所描述的）指出，从按值传递转换到按引用传递，我们只需要做一件事情，即在类型名和参数名之间插入&符号。不需要在函数体中或者在函数调用的语法中做任何其他修改。下面是示例：

```
void read (ifstream& f, Person* &person)     // read one record
{ char buf[80];
  Person* p = new Person;                    // allocate space for new object
  . . .                                      // the rest of read()
  person = p; }                              // hook up new object
```

正是由于我们一开始将这个参数命名为Person* person而不是Person *person，所以这种转换才更加容易。但这并不重要，在这种情况下，C++忽略空格，我们可以按任何看起来合适的方法在类型名和参数名之间对齐星号（*和&符号）。

那么按指针传递指针又如何呢？这也没问题。但要注意程序中的三个位置：函数调用、函数接口与函数体。在函数调用时，要在变量名前（如在指针名data[cnt]前）插入&。下面就是这个函数调用的样子：

```
while (!from.eof())                    // read until eof
{ read(from, &data[cnt]);              // passing pointer by pointer
  cnt++; }
```

在函数接口中，要在参数名前插入*，即应该用Person* *person（或Person** person）而不是Person* person。

在函数体中是最容易出问题的，应在参数名前使用*。该参数名为person，而不是*person或**person。因此，read()函数中的最后一行应进行间接引用。

```
void read (ifstream& f, Person** person)     // pointer by pointer
{ char buf[80];
```

```
    Person *p = new Person;                    // allocate space for new object
    . . .                                      // the rest of read()
    *person = p; }                             // bingo!
```

这根本不算难，但按引用传递指针比按指针传递指针要简单得多。这对任意类型也是适用的，按引用传递更为简单，而且不那么容易出错。

至此为止，我们已经讨论了把数据输入到计算机中数组的技术。这些讨论和动态绑定、多态性以及其他论题都没有关系。我们讨论这些只是因为需要一个运行的程序，而且复习一下第6章和第7章中关于动态内存管理、文件I/O和参数传递的内容并没有什么坏处。

当程序开始处理已经在内存中的数据时就有动态绑定的问题了。由于要以不同的方式处理不同种类的对象，程序不得不识别每个函数调用中所处理的对象的类型。因而要在Person类中使用kind域。在这个简单的例子中，“处理数据”指的是遍历指针数组，打印每个Person对象，或者作为教师（显示rank域）或者作为学生（显示major域）。在实际生活中会有一些函数不得不以不同的方式处理各种对象。在这个例子中，我们将这个处理任务放在main()函数中。

```
for (int i=0; i < cnt; i++)                    // go over the array of pointers
{ cout <<" id:    " <<data[i]->id <<endl;      // print id, name
  cout <<" name:  " <<data[i]->name <<endl;
  if (data[i]->kind == 1)
    cout <<" rank:  " <<data[i]->rank <<endl;  // faculty rank
  else if (data[i]->kind == 2)
    cout <<" major: " <<data[i]->major <<endl; // student major
  cout << endl; }
```

该循环是程序的核心：它根据对象的实际类型对异构对象列表进行处理。首先，无条件地处理对所有类型的对象都要处理的事——以合适的标题格式打印学校代码和名字。然后根据对象的类型处理每个对象，该循环访问对象的kind域并决定打印职称还是专业。

图15-9是处理输入文件的过程，其完整程序见程序15-3。除了前面描述的Person类型和read()函数外，这个程序还包含了扮演客户代码角色的main()函数，该函数中定义了指针数组和文件对象，在循环中读取输入数据并进行处理，然后返回动态内存。这个程序的执行结果如图15-11所示。

程序15-3　用传统的方法处理异构对象列表

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
       int kind;                           // 1 for faculty, 2 for student
       char id[10];                        // fixed length
       char* name;                         // variable length
       char* rank;                         // for faculty only
       char* major;                        // for student only
   } ;

void read (ifstream& f, Person*& person)   // read one record
{ char buf[80];
  Person* p = new Person;                  // allocate space for new object
  f.getline(buf,80);                       // recognize the incoming type
```

```
  if (strcmp(buf, "FACULTY") == 0)
       p->kind = 1;                              // 1 for faculty
  else if (strcmp(buf, "STUDENT") == 0)
       p->kind = 2;                              // 2 for student
  else
       p->kind = 0;                              // type not known
  f.getline(p->id,10);                           // read id
  f.getline(buf,80);                             // read name
  p->name = new char[strlen(buf)+1];             // allocate space
  strcpy(p->name, buf);                          // copy name
  f.getline(buf,80);                             // read rank/major
  if (p->kind == 1)
  { p->rank = new char[strlen(buf)+1];           // space for rank
    strcpy(p->rank, buf); }                      // copy rank
  else if (p->kind == 2)
  { p->major = new char[strlen(buf)+1];          // space for major
    strcpy(p->major, buf); }                     // copy major
  person = p;                                    // hook it up to array
}

int main()
{
  Person* data[20]; int cnt = 0;                 // array of pointers
  ifstream from("univ.dat");                     // input data file
  if (!from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                       // read until eof
    cnt++; }
  cout << " Total records read: " << cnt << endl << endl;
  for (int i=0;  i < cnt;  i++)
  { cout <<" id:     " <<data[i]->id <<endl;     // print id, name
  cout <<" name:   " <<data[i]->name <<endl;
  if (data[i]->kind == 1)
    cout <<" rank:   " <<data[i]->rank <<endl;  // faculty rank
  else if (data[i]->kind == 2)
    cout <<" major: " <<data[i]->major <<endl; // student major
  cout << endl; }
for (int j=0;  j < cnt;  j++)
{ delete [] data[j]->name;                      // delete name
  if (data[j]->kind == 1)
    delete [] data[j]->rank;                    // delete rank/major
  else if (data[j]->kind == 1)
    delete [] data[j]->major;
  delete data[j]; }                             // delete the record
return 0;
}
```

这个解决方法没有任何困难和令人困惑的地方（指针符号可能是个例外）。虽然我们使用了许多很好的C++语言的修饰（例如，结构、指针、用运算符new和delete进行动态内存管理、按引用传递参数、库文件对象等），这个程序也可用其他任何语言编写。它实现了动态绑定的目的（根据每个对象的类型来处理它们）。但是它没有利用语言中面向对象的特点（如将数据和操作绑定在一起、构造函数和析构函数、将任务推向服务器、将应该属于一起的信息放在一起以及继承等）。

```
Total records read: 4

id:  U12345678
name: Smith, John
rank: Associate Profesor

id:  U12345611
name: Jones, Jan
major: Computer Science

id:  U12345689
name: Black, Jeanne
rank: Assistant Profesor

id:  U12345622
name: Green, James
major: Astronomy
```

图15-11　对异构对象列表处理的输出结果

15.3.2　动态绑定：面向对象的方法

在下面的版本中，我们将创建三个类：Person、Faculty和Student类。Faculty和Student相同的所有特征都放在基类Person中，即id、name、kind域。我们不再使用数值转换来表示对象的类型（1表示教师，2表示学生），而是引入有自解释性的值的枚举类型。当有几种不同种类的对象，而且可以增加新的对象时，使用枚举类型尤其方便。

我们将基类Person的数据成员定义为受保护的而不是私有的，这样派生类Faculty和Student可以比较容易地访问这些数据成员，而不用麻烦Person类的设计人员提供微不足道的访问函数。

```
struct Person {
public:
  enum Kind { FACULTY, STUDENT } ;
protected:
  Kind kind;                          // FACULTY or STUDENT
  char id[10];                        // data common to both types
  char* name;                         // variable length
public:
  Person(const char id[], const char nm[], Kind type);
  Kind getKind() const;
  ~Person(); } ;
```

构造函数接收三个参数来初始化对象的三个数据成员。它执行在前一版本中函数read()函数执行的那些操作，即为名字动态分配内存。析构函数执行前一版本中的函数main()执行的那些操作，即回收堆内存。这个例子虽然很小，但是很好地演示了将本来可以分开来放置在代码不同部分（这样会增加协调的需要）的信息放在一起。

另一个成员函数getKind()是一个辅助函数，这个消息由客户代码（read()函数）传送给派生类（Faculty和Student类）的对象，辨别对象的种类。在前面的版本中，read()函数直接访问kind域，代码的不同部分之间存在着依赖。在这个设计中，kind域定义为受保护的，不是公共的，Person类必须为其派生类的客户提供访问函数。尚不能肯定

这是对直接访问kind域的很大改进，但是这在实践中常常用到（而且可能会有一些改进）。

我们对待客户代码要比对待派生类更加谨慎。对于派生类，我们心安理得地允许它直接访问基类的数据成员。对于客户代码，尽管很不情愿，我们还是提供了访问服务器数据成员的函数。

派生类Faculty和Student公共地继承Person类。尽管使用了struct关键字来定义它们，我们仍然显式地指定继承模式为公共继承以避免混淆。如果未指明继承模式，缺省也是公共模式。公共模式是最自然和最方便的，它不会从派生类的客户程序中去掉基类的功能。在本例中这不是很重要，因为基类很小，客户代码只能使用一个功能即getKind()方法。

然而，在这里使用公共继承是很重要的。我们使用了Person类型的指针的数组，但是要将这些指针指向Faculty和Student类的对象。这涉及到转换，为了进行隐式转换，C++要求继承模式必须为公共继承。

这当然也是方便性的问题，如果可能就要使用显式转换。但是还有另外一个更加重要的原因。我们最终要在这个设计中使用虚函数，它允许客户代码调用派生类的方法，例如write()，并让运行时系统分辨出这个函数属于哪个派生类。只有在派生模式为公共时这种行为才有可能发生。

```
struct Faculty : public Person {                    // public inheritance
private:
  char* rank;                                       // for faculty only
public:
  Faculty(const char id[], const char nm[], const char r[]);
  void write () const;                              // display record
  ~Faculty(); } ;                                   // return heap memory

struct Student : public Person {                    // public inheritance
private:
  char* major;                                      // for student only
public:
  Student(const char id[], const char nm[], const char m[]);
  void write () const;                              // display record
  ~Student(); } ;                                   // return heap memory
```

派生类Faculty和Student从基类Person中继承了所有的数据成员，并根据Person的种类（rank或major）定义了它们各自的数据。

派生类Faculty和Student的构造函数接受初始化所有域所必需的参数，无论这些域是在派生类中定义的还是从基类Person中继承而来的。派生类的构造函数应把数据传递给初始化列表中的基类构造函数。正如我们在Person类的构造函数接口中看到的，它包含了正在创建对象的种类（Faculty或Student）的说明。对许多程序员而言，这意味着派生类构造函数的参数列表中不仅应该包含用于初始化基类部分的数据（三个参数），还应包含用于初始化派生类部分的数据（Student的为major，Faculty的为rank）。

```
Faculty(const char id[], const char nm[], Kind k, const char r[])
   : Person(id,nm,k)                                // initialization list
 { rank = new char[strlen(r)+1];
   if (rank == 0) { cout << "Out of memory\n"; exit(0); }
   strcpy(rank,r); }
```

这是将任务交给派生类客户的典型例子。客户代码（read()函数）将以如下形式创建

Faculty对象：

```
person = new Faculty(id,name,FACULTY,buf); }    // object is Faculty
```

但这是不合理的要求，客户代码已说明它创建了一个Faculty对象，为什么还要强迫它做无用的工作，传递一个参数来说明是Faculty呢？这个任务应交给Faculty对象，它应该知道自己是Faculty，而且应该把这个信息告诉它的Person部分，而没有必要将read()函数拖到需要协调工作的循环中。

```
Faculty(const char id[], const char nm[], const char r[])
       : Person(id,nm,FACULTY)      // this is what OOP is all about
 { rank = new char[strlen(r)+1];
   if (rank == 0) { cout << "Out of memory\n"; exit(0); }
   strcpy(rank,r); }
```

现在，客户代码的任务就很简单了：

```
person = new Faculty(id,name,buf); }      // object is Faculty
```

注意，在Faculty和Student类的定义中，构造函数原型只有3个参数，而不是4个。这是面向对象程序设计的本质：寻找合适的方法在协作类之间分配任务。

派生类的析构函数回收由构造函数分配的堆内存（Faculty类为rank，Student类为major）。

在两个派生类中都实现了write()成员函数，这两个函数非常相似，因此有相同的函数名。这就要求在基类中实现write()函数。但由于不同种类的Person对象的算法不同，因此让每个派生类都分别定义各自的函数，并使用相同的函数名，因此这些函数是多态的。

write()函数实现了前面版本中的main()函数的部分功能。由于每个派生类Faculty和Student都知道各自的特性，因而没有必要研究目标对象的种类。

```
void Faculty::write () const                          // display record
{ cout << " id:     " << id << endl;                  // print id, name
  cout << " name:   " << name << endl;
  cout << " rank:   " << rank <<endl <<endl; }        // faculty only

void Student::write () const                          // display record
{ cout << " id:     " << id << endl;                  // print id, name
  cout << " name:   " << name << endl;
  cout << " major:  " << major <<endl <<endl; }       // student only
```

全局函数read()是前一版本的程序中该函数的精简形式。它从输入文件中把数据读取到局部数组中，然后检查kind[]数组来决定创建哪种类型的对象。如果是“FACULTY”，read()函数创建一个新的Faculty对象（使用运算符new）。如果是“STUDENT”，则read()函数创建一个新的Student对象（还是使用运算符new）。在这两种情况下，数据都作为参数传递给类的构造函数。

```
void read (ifstream& f, . . . ?? person)          // what is its type?
{ char kind[8], id[10], name[80], buf[80];
  f.getline(kind,80);                             // recognize the incoming type
  f.getline(id,10);                               // read id
  f.getline(name,80);                             // read name
  f.getline(buf,80);                              // rank or major?
  if (strcmp(kind, "FACULTY") == 0)
```

```
{ person = new Faculty(id,name,buf); }          // object is Faculty
else if (strcmp(kind, "STUDENT") == 0)
{ person = new Student(id,name,buf); }          // object is Student
else
{ cout << " Corrupted data: unknown type\n"; exit(0); } }
```

那么，这个函数的第二个参数应该是什么类型呢？应该是一个指针。否则，它将不能接收new操作的返回值。而且这个参数应按引用传递而不是按值传递；否则，只能由局部指针person指向这个新创建的对象，而不能由实际参数指向这个对象，这样就不能在客户代码中访问这个对象了。同时，这个参数不能是Faculty指针，否则不能指向Student对象；也不能是Student指针，否则不能指向Faculty对象。

这样，它既不能是Faculty指针也不能是Student指针，那么应该是什么呢？什么类型的指针才可以指向不同类类型的对象呢？回顾一下本章的第一节，如果不同的类不是通过继承相关的，就没有指针可以指向这些类的对象，并且完成任何有意义的工作。再回顾一下本章第二节，如果这些不同类之间存在着继承关系，那么基类指针可以指向任意一个派生类的对象，A、B或者其他的对象。只要目标对象所属的类是在继承层次的限制之内，一个强大的指针就可以指向任何对象。

因此，这个参数必须是Person类的指针，与前面版本中一样。在read()函数内，创建不同派生类的对象，并由基类的指针指向这些对象。

```
void read (ifstream& f, Person*& person)        // read one record
{ char kind[8], id[10], name[80], buf[80];
  f.getline(kind,80);                           // recognize the incoming type
  f.getline(id,10);                             // read id
  f.getline(name,80);                           // read name
  f.getline(buf,80);                            // rank or major?
  if (strcmp(kind, "FACULTY") == 0)
  { person = new Faculty(id,name,buf); }        // object is Faculty
  else if (strcmp(kind, "STUDENT") == 0)
  { person = new Student(id,name,buf); }        // object is Student
  else
  { cout << " Corrupted data: unknown type\n"; exit(0); } }
```

在main()中以与前一版本中相同的方法调用这个read()函数。其区别是data[]数组的（Person*类型的）成员现在指向不同派生类（Faculty或Student）的对象。

```
int main()
{ cout << endl << endl;
  Person* data[20]; int cnt = 0;                // array of pointers
  ifstream from("univ.dat");                    // input data file
  if (!from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                      // read until eof
    cnt++; }
  . . . }                                       // the rest of main()
```

但问题是基类指针是短视的，它不能激活在派生类中定义的操作。这不是一个难题。只要基类指针指向派生类对象，总有方法可以告诉编译程序我们知道基类指针指向的是派生类对象。这个方法就是使用到派生类的转换。

我们将这个决策封装在一个函数中，例如write()。如果操作对相似对象所属的不同

类型执行了不同的任务，就应该为每个操作设计类似的决策函数。这里的困难在于决定这个函数的参数类型。下面是这个函数的轮廓。

```
void write (  . . . ??  p)                    // display record
{ switch (p.getKind()) {                      // get object type
     case Person::FACULTY:
          . . .;  break;                      // do it Faculty way
     case Person::STUDENT:
          . . .;  break;  } }                 // do it Student way
```

这个函数的参数应该是什么类型呢？它可以接受两种类型的实际参数：Faculty对象和Student对象。如果对象类型是Faculty类，则这个函数只调用Faculty::write()。如果参数类型是Student类，则只调用Student::write()。

在这里应该再次使用前一节中关于类之间转换机制的知识。我们可以试图用Person类型的参数，因为Faculty对象和Student对象都可拷贝到Person对象中（派生类对象有足够多的数据初始化基类对象）。

```
void write (Person p)                         // display record
{ switch (p.getKind()) {                      // get object type
     case Person::FACULTY:
          . . .;  break;                      // do it Faculty way
     case Person::STUDENT:
          . . .;  break;  } }                 // do it Student way
```

这个方法的第一个问题是按值传递参数，当对象要管理动态内存时这显然不是一个好的做法。第二个问题是函数体内全是Person对象，没有办法将基类对象转换回派生类对象。原始数据被截取了，不能恢复。即使我们在每个派生类中增加一个构造函数，该构造函数将基类对象转换为派生类对象（如程序15-2中那样），这仍不够。这个构造函数所能做的最主要是为派生类中定义的域设置缺省值。但我们所需的是教师职称和学生专业的原始值。

这就可以排除使用基类对象值作为函数参数的可能性，但并不排除使用基类指针作为函数的参数。派生类指针（可以执行派生类中所有操作的强大的、远视的指针）可以比较容易地转换为基类指针，这是一种安全转换，因而不需要进行显式转换。这样没有截取数据，转换回派生类指针也是可能的。（但转换回派生类指针是不安全的转换，需要显式转换）。

```
void write (Person* p)                        // display record
{ switch (p->getKind()) {                     // get object type
     case Person::FACULTY:
          . . .;  break;                      // do it Faculty way
     case Person::STUDENT:
          . . .;  break;  } }                 // do it Student way
```

这个转换过程中丢失了执行派生类中定义的操作的能力。这个弱小的、短视的基类指针所能做的只是访问在基类中定义的函数。但这种方法的主要优点是基类指针仍指向派生类的对象。在switch语句中，write() 函数只是区别实际参数指向的是Faculty对象还是Student对象。因此，剩下的事是调用Faculty类的write()方法或Student类的write()方法。

```
void write (const Person* p)                  // display record
{ switch (p->getKind()) {                     // get object type
     case Person::FACULTY:
```

```
        p->write();  break;                    // do it Faculty way
    case Person::STUDENT:
        p->write();  break; } }                // do it Student way
```

由于指针p是一个基类指针，它只能访问基类方法。因此，在switch语句的两个分支中的write()函数调用或者访问基类的write()函数（如果Person类有write()方法），或者导致语法错误（如果Person类没有write()方法）。

正如我们可以看到的，这个write()函数刚刚了解了其参数指针所指向的对象的种类。但编译程序并不知道这些，它只知道这是Person类的指针。因而函数write()应该告诉编译程序它所知道的信息，即应该将基类指针转换为Faculty类（第一个switch分支）或Student类（第二个switch分支）。

```
void write (const Person* p)                   // display record
{ switch (p->getKind()) {                      // get object type
    case Person::FACULTY:
        ((Faculty*)p)->write();  break;        // do it Faculty way
    case Person::STUDENT:
        ((Student*)p)->write();  break;        // do it Student way
} }
```

正如许多转换那样，这些转换看起来既糟糕又复杂。但是它们确实做了我们刚才所描述的事情，即将（类Person*的）指针p转换成了Faculty*类型的指针（第一个switch分支）或Student*类型的指针（第二个switch分支）。括号尽管很讨厌，但是应该使用，因为箭头选择运算符的操作比转换的操作级别高。如果省略了括号，例如使用(Faculty*)p->write()，编译程序会认为我们希望转换write()调用的返回值，而不是转换指针p。因此还是要保留这些括号。

在循环中调用write()函数，接收指向Faculty对象或Student对象的Person指针作为实际参数。

```
for (int i=0;  i < cnt;  i++)
  { write(data[i]); }                          // display data
```

该完整程序见程序15-4。

程序15-4 用面向对象方法处理异构对象列表

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
public:
       enum Kind { FACULTY, STUDENT } ;
protected:
       Kind kind;                              // FACULTY or STUDENT
       char id[10];                            // data common to both types
       char* name;                             // variable length
public:
       Person(const char id[], const char nm[], Kind type)
       { strcpy(Person::id,id);                          // copy id
         name = new char[strlen(nm)+1];                  // get space for name
         if (name == 0) { cout << "Out of memory\n"; exit(0); }
```

```
        strcpy(name,nm);                                    // copy name
        kind = type; }                                      // remember its type

    Kind getKind() const
      { return kind; }                                      // access Person's type

      ~Person()
      { delete [] name; }                                   // return heap memory
} ;

struct Faculty : public Person {
private:
      char* rank;                                           // for faculty only

public:
      Faculty(const char id[], const char nm[], const char r[])
        : Person(id,nm,FACULTY)                             // initialization list
      { rank = new char[strlen(r)+1];
        if (rank == 0) { cout << "Out of memory\n"; exit(0); }
        strcpy(rank,r); }

  void write () const                                       // display record
  { cout << " id:     " << id << endl;                      // print id, name
    cout << " name:   " << name << endl;
    cout << " rank:   " << rank <<endl <<endl; }            // faculty only

      ~Faculty()
      { delete [] rank; }                                   // return heap memory
} ;

struct Student : public Person {
private:
      char* major;                                          // for student only

public:
      Student(const char id[], const char nm[], const char m[])
        : Person(id,nm,STUDENT)                             // initialization list
      { major = new char[strlen(m)+1];
        if (major == 0) { cout << "Out of memory\n"; exit(0); }
        strcpy(major,m); }

  void write () const                                       // display record
  { cout << " id:     " << id << endl;                      // print id, name
    cout << " name:   " << name << endl;
    cout << " major:  " << major <<endl <<endl; }           // student only
  ~Student()
  { delete [] major; }                                  // return heap memory
} ;

void read (ifstream& f, Person*& person)                // read one record
{ char kind[8], id[10], name[80], buf[80];
  f.getline(kind,80);                                   // recognize the incoming type
  f.getline(id,10);                                     // read id
  f.getline(name,80);                                   // read name
  f.getline(buf,80);                                    // rank or major?
  if (strcmp(kind, "FACULTY") == 0)
  { person = new Faculty(id,name,buf); }                // object is Faculty
  else if (strcmp(kind, "STUDENT") == 0)
```

```
    { person = new Student(id,name,buf); }          // object is Student
    else
    { cout << " Corrupted data: unknown type\n"; exit(0); }
}

void write (const Person* p)                        // display record
{ switch (p->getKind()) {                           // get object type
     case Person::FACULTY:
          ((Faculty*)p)->write();  break;           // do it Faculty way
     case Person::STUDENT:
          ((Student*)p)->write();  break;           // do it Student way
} }

int main()
{ cout << endl << endl;
  Person* data[20]; int cnt = 0;                    // array of pointers
  ifstream from("univ.dat");                        // input data file
  if (!from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                          // read until eof
    cnt++; }
  cout << " Total records read: " << cnt << endl << endl;
  for (int i=0;  i < cnt;  i++)
  { write(data[i]); }                               // display data
  for (int j=0;  j < cnt;  j++)
  { delete data[j]; }                               // delete the record
  return 0;
  }
```

这种方法比前一种方法好得多。数据和操作绑定在一起，任务推向服务器类，并避免将相关代码拆开。和所有的面向对象方法一样，源代码比相应的非面向对象方法的代码要长。除此之外，该程序和程序15-3完成的工作一样。它的输出结果与程序15-3的输出结果也相同（见图15-11）。

下面我们要做的事是去掉write()函数中对对象种类的测试。不再测试目标对象的种类，而是将指针转换回派生类指针再激活合适的派生类函数，我们希望编译程序完成所有这些工作。编译程序应生成测试对象类型的目标代码，执行转换，调用合适的方法。其关键是在指定基类成员函数时使用virtual关键字。

15.3.3 动态绑定：使用虚函数

关键字virtual是一种语法技巧。它为传递给派生类对象的消息创建运行时的类型解析属性。为了使用这个属性，我们在基类和每个派生类中实现有同一名字的函数。

在我们讨论的例子中，这意味着要为基类Person及派生类Faculty和Student实现write()方法，可以用如下形式编写全局的write()函数。

```
void write (const Person* p)                        // display record
{ p->write(); }                                     // is not this nice?
```

对于编译时绑定，这只是意味着调用在Person类中定义的write()方法。对于运行时绑定，编译程序所生成的目标代码将分析基类指针p所指向的对象的类型，并确定调用的方法属于哪种类型，再从那种类型中调用write()方法。根据指针所指向的对象，将调用

Faculty类的方法或Student类的方法。所有这些都是在运行时定义的。

但是实现这种方法有一些限制。属于派生类的虚函数只能由基类指针或基类引用激活。如果将消息传递给对象，无论是基类对象还是派生类对象，都不能进行运行时绑定，而要使用静态绑定：无论对象的类型是什么，这个类的消息都将能激活。

例如，x.write()的意义依赖于对象x的类型，这个类型是在编译时指定的，而不是在执行时。

虚函数不能是静态函数，不能通过类作用域运算符调用虚函数，只能由指向派生类对象的基类指针（引用）调用。

继承的模式必须为public，不能是protected或private。隐式转换只有在公共派生时才有效。

将继承层次中基类的成员函数指定为虚函数后，每个派生类都必须实现与该虚函数同名的函数。派生类中对函数重定义时，要求函数名字、标识、返回类型都必须与基类的虚函数完全一致。

如果在派生类中的函数名不同也没有问题，但这个函数不能称为使用了运行时绑定。动态绑定的基础是使用相同的函数，但有不同的解释。

如果函数标识不同，派生类中的方法将隐藏基类方法，从而破坏了虚函数机制。例如，如果派生类中定义了一个不带参数返回值为空的函数write()，基类定义了一个void函数write(int)，就不能使用动态绑定调用派生类函数。这种情况下，p->write()将激活指针p所属类的函数。如果类中有这个函数则调用，如果没有会出现语法错误。

如果虚函数的返回值与派生类的不同，也会出现语法错误，即使函数的标识是相同的。

关键字virtual只在基类的类定义中出现，没有必要在基类的函数定义中重复出现，也没有必要在派生类定义中重复出现。

如果继承层次不止两层，可以在任何层次定义虚函数。没有强制一定要在层次的最上面，或在每个继承的下一级层次中实现被定义的函数。虚函数可以间接继承。

如果满足了所有的要求，就没有必要在基类中定义kind域了，当然也没有必要定义返回kind域值的方法。将程序15-3转换为一个带有虚函数的程序，所要做的就是在Person类中定义write()函数，该函数的返回类型必须为void并且没有参数。

```
struct Person {
protected:
  char id[10];                                    // no Kind
  char* name;
public:
  Person(const char id[], const char nm[]);       // no Kind
  virtual void write () const;                    // const is part of signature
  ~Person(); } ;
```

这样，就没有必要将派生类的kind信息交给其基类了。

```
struct Faculty : public Person {
private:
       char* rank;                                          // for faculty only
public:
       Faculty(const char id[], const char nm[], const char r[])
         : Person(id,nm)                                    // no FACULTY
       { rank = new char[strlen(r)+1];
```

```
        if (rank == 0) { cout << "Out of memory\n"; exit(0); }
        strcpy(rank,r); }
  void write () const                                    // it is virtual now
  { cout << " id:     " << id << endl;                   // print id, name
    cout << " name:   " << name << endl;
    cout << " rank:   " << rank <<endl <<endl; }         // faculty only
      ~Faculty()
      { delete [] rank; } } ;                            // return heap memory
```

更重要的是，没有必要在客户代码中检查对象的类型，程序15-5从程序15-4修改而来，它使用虚函数write()去掉了客户代码中的子类型分析。程序的输出结果与前面版本相同（见图15-11）。

程序15-5 使用虚函数处理异构列表

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
protected:
      char id[10];                                  // data common to both types
      char* name;                                   // variable length

public:
      Person(const char id[], const char nm[])  //, Kind type)
      { strcpy(Person::id,id);                      // copy id
        name = new char[strlen(nm)+1];              // get space for name
        if (name == 0) { cout << "Out of memory\n"; exit(0); }
        strcpy(name,nm);                            // copy name
      }

      virtual void write() const                    // not much to do
      { }

      v~Person()                                    // return heap memory
      { delete [] name; }                           // for Person object only
} ;
struct Faculty : public Person {
private:
      char* rank;                                   // for faculty only

public:
      Faculty(const char id[], const char nm[], const char r[])
        : Person(id,nm)                             // initialization list
      { rank = new char[strlen(r)+1];
        if (rank == 0) { cout << "Out of memory\n"; exit(0); }
        strcpy(rank,r); }

  void write () const                               // display record
  { cout << " id:     " << id << endl;              // print id, name
    cout << " name:   " << name << endl;
    cout << " rank:   " << rank <<endl <<endl; } // faculty only

      ~Faculty()
      { delete [] rank; }                           // return heap memory
} ;
```

```
struct Student : public Person {
private:
       char* major;                                  // for student only

public:
       Student(const char id[], const char nm[], const char m[])
         : Person(id,nm)                             // initialization list
       { major = new char[strlen(m)+1];
         if (major == 0) { cout << "Out of memory\n"; exit(0); }
         strcpy(major,m); }

  void write () const                                // display record
  { cout << " id:     " << id << endl;               // print id, name
    cout << " name:   " << name << endl;
    cout << " major: " << major <<endl <<endl; }// student only

  ~Student()
  { delete [] major; }                               // return heap memory
} ;

void read (ifstream& f, Person*& person)             // read one record
{ char kind[8], id[10], name[80], buf[80];
  f.getline(kind,80);                                // recognize the incoming type
  f.getline(id,10);                                  // read id
  f.getline(name,80);                                // read name
  f.getline(buf,80);                                 // rank or major?
  if (strcmp(kind, "FACULTY") == 0)
  { person = new Faculty(id,name,buf); }             // object is Faculty
  else if (strcmp(kind, "STUDENT") == 0)
  { person = new Student(id,name,buf); }             // object is Student
  else
  { cout << " Corrupted data: unknown type\n"; exit(0); }
}

void write (const Person* p)                                 // display record
{ p->write(); }                                              // Faculty or Student?

int main()
{ cout << endl << endl;
  Person* data[20]; int cnt = 0;                             // array of pointers
  ifstream from("univ.dat");                                 // input data file
  if (!from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                                   // read until eof
    cnt++; }
  cout << " Total records read: " << cnt << endl << endl;
  for (int i=0;  i < cnt;  i++)
  { write(data[i]); }                                        // display data
  for (int j=0;  j < cnt;  j++)
  { delete data[j]; }                                        // delete the record
  return 0;
  }
```

多态性（运行时对消息对象进行解释）的基础是将派生类对象隐式转换为基类对象的合法性：基类指针（例子中的Person）可以不使用显式转换而指向派生类的对象（Faculty或Student）。

```
Person *p, *pf, *ps;                          // pointers of type Person
p = new Person("U12345678", "Smith");
pf = new Faculty("U12345689", "Black", "Assistant Professor");
ps = new Student("U12345622", "Green", "Astronomy");
```

显式转换是可选的，可以用来让维护人员注意到指针之间的类型转换。

```
ps = (Person*) new Student("U12345622", "Green", "Astronomy");
```

使用虚函数后，就没有必要将消息转换回派生类指针指向的对象类型。注意，指向派生类对象的指针必须使用显式转换才能指向基类对象，派生类指针也必须使用显式转换才能访问基类方法。

```
Student* s = (Student*)ps;          // cast is mandatory
s->write();                         // derived class pointer
```

虚函数使得基类指针可以调用派生类的成员函数。

```
ps->write();              // base class pointer
```

然而，所有这些改善都是装饰性的，只是改善了客户代码的外貌而已。实际上，程序15-5和程序15-4做的工作是完全一样的。kind域从Person类中消失了，但是实际上它仍然存在，只不过是由编译程序产生的代码访问而不是被程序员写的源代码访问。条件语句也从客户代码中消失了，但是它也仍然存在，也是由编译程序产生的代码实现而不是由程序员写的源代码实现。

程序15-4显式地分配了额外的空间来分析Person对象的类型，还花费了额外的时间判断要调用哪一个write()函数。程序15-5也花费了同样额外的空间和时间。

一些程序员，特别是那些编写实时控制系统的程序员会认为虚函数是浪费。这是不公平的，因为消耗额外空间和时间的是多态性算法。不论是如程序15-4那样显式地实现，还是如程序15-5那样使用虚函数来实现，几乎没有什么不同。

但是，许多程序员喜欢虚函数而且将任何事情都看做虚的。不论是否使用多态性算法，虚函数都会消耗一些空间和执行时间。如果在我们的应用程序中这些资源很稀少，应该确保只定义那些让客户代码简单的函数为虚函数。

15.3.4 动态绑定与静态绑定

动态绑定为程序员提供了一种新的、更激动人心的方式处理算法。我们可以在某个通用的公共基类下创建许多相关的派生类，每个派生类中有一个函数以各自的方式进行不同的处理，但这些函数的名字与接口必须相同，然后通过基类指针调用这些函数。被调用的函数不依赖于指向对象的指针类型，而是依赖于被指针所指向的对象类型。真是太棒了！

但有了动态绑定并不意味着传统的静态绑定无关紧要。在大多数C++程序设计中，调用的方法依赖于指向对象的指针类型，而不是指针所指向的对象类型。这引入了另外一些复杂性问题。

对于静态绑定，在分析函数调用时，必须考虑消息的目标对象的类型及被调用方法的标识。当动态绑定有可能时，还必须考虑几个其他的因素。

首先，要考虑消息的目标是一个对象还是一个指针（或引用）。如果是一个对象，只可能是静态绑定，我们所要考虑的只是函数的标识以确定这个函数调用是否正确。如果消息的目

标是一个指针或引用，则有可能使用动态绑定。

其次，要定义指针属于继承层次中的哪个位置。如果指针属于基类，则有可能是动态绑定——这依赖于指针指向的对象类型及函数定义的方式。如果指针属于某个派生类，则只可能是静态绑定。然而，调用的结果也要依赖于指针指向的对象类型及函数定义的方法。

这样我们就需要考虑两个因素：指针指向的对象类型及函数定义的方法。这个对象可能属于基类类型（不能动态绑定）和某个派生类（只有当对象是被基类指针指向时，才可能是动态绑定）。函数可能在基类中定义或在派生类中定义，而且函数也可以同时在基类和派生类中定义。这种情况下，要区别派生类中重定义的函数，看其标识是否与基类中定义的一致。只有以相同标识重定义的才能支持动态绑定。否则只能支持静态绑定，而且不一定都能被某给定的指针类型和对象类型组合调用。总之，要区别以下四种不同的成员函数：

- 在基类中定义，并在派生类中继承而没有进行重定义的函数。
- 在派生类中定义，而在基类中没有与之相应部分的函数。
- 在基类中定义，并在派生类中使用相同的函数名和不同的函数标识重定义了的函数。
- 在基类中定义，并在派生类中使用与虚函数相同的函数名和相同的函数标识重定义的函数。

这看起来很复杂吗？是的，尤其是初次学习虚函数时。但很快会变得简单起来。

对于指向基类对象的基类指针，只能调用在基类中定义的方法，无论它们是否由派生类继承或重新定义。试图调用在派生类中定义而在基类中没有的函数会产生语法错误。调用派生类中重定义的函数是徒劳的——但可以以任意方式调用基类中定义的函数。

对于指向派生类对象的派生类指针（指针和对象属于同一类），不能调用基类的函数，除非这些定义在基类中的函数被原样继承下来。该指针可以调用在派生类中增加的方法和重定义的方法。在派生类中重定义的函数是被静态调用的，与如何定义没有关系，无论它们是否使用相同的标识，是否作为虚函数。

注意，在派生类中重定义的基类函数不能被指向派生类对象的派生类指针访问；它们被相应的派生类函数隐藏。试图去访问这种基类函数时，如果函数标识一致会导致静态调用在派生类中定义的函数（不论是否是虚函数）；如果标识不一致则出现语法错误。

指向派生类对象的基类指针可以访问这个派生类继承的（并没有重定义的）所有基类方法，但不能访问在派生类中定义的而在基类中没有的函数。如果派生类重新定义某个基类方法为非虚函数（无论是否有相同的标识），这个派生的方法也不能通过基类指针访问——而是静态调用基类中相应的方法。如果派生类重新定义某个基类方法为虚函数时，通过基类指针访问的是这个派生类方法，而不是基类的方法。这才是可能使用动态绑定的惟一情况。

指向基类对象的派生类指针不像常规那样，它可以调用在基类中定义的方法及从基类继承下来没有重新定义的方法，而不能调用在派生类中重新定义的基类方法，这些方法对该指针而言是被隐藏的。它也不能调用重新定义基类方法的派生类方法（不论是否作为虚函数，也不论是否使用相同的标识）。基类对象不支持这些方法，这样做会导致运行时错误：崩溃或产生不正确的结果。

上面这段描述很长也很繁锁，但其基础是两条简单的基本原则：

- 指向派生类对象的派生类指针可以访问派生类中定义的方法和从基类原样继承而来的方

法。对派生类的指针而言，在派生类中重定义的方法隐藏了基类中定义的（有相同或不同函数标识，或者为虚函数或者不为虚函数的）方法。

- 指向派生类对象的基类指针只能访问在基类中定义的方法，但有一个例外：如果某个函数在派生类中重定义为虚函数，则基类指针使用动态绑定所激活的是派生类函数，而不是基类函数。

这很简单，但是可能需要一些时间去理解和消化。在编写本书时，我们努力构造一些简单的图形或者表格来表示这些讨论结果，如图15-12和表15-1所示。

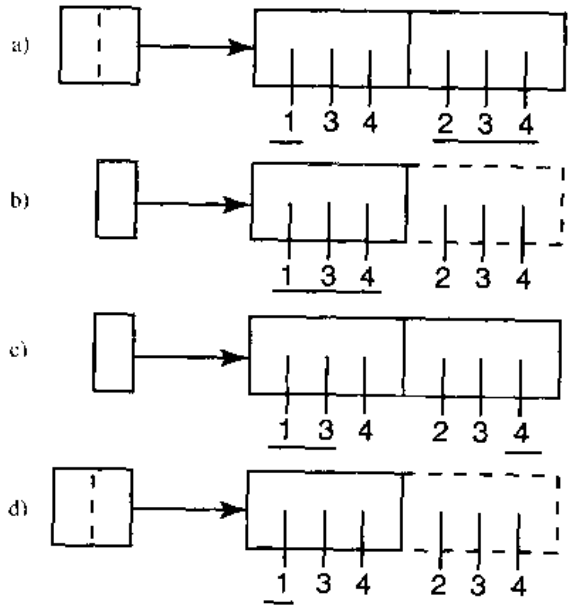


图15-12 基类指针和派生类指针的静态绑定与动态绑定

表15-1 静态绑定与动态绑定的规则总结

成员函数的种类	基类指针		派生的指针	
	基类对象	派生的对象	基类对象	派生的对象
在基类中的函数定义				
在派生的类中继承	有效的	有效的	有效的	有效的
在派生类中重定义（非虚）	有效的	有效的	无效的	隐藏
在派生类中重定义（虚）	有效的	隐藏	无效的	隐藏
在派生类中的函数定义				
只在派生的类中定义	语法错误	语法错误	crash	隐藏
在派生类中重定义（非虚）	无效	无效	crash	隐藏
在派生类中重定义（虚）	无效	动态绑定	crash	隐藏

图15-12表示了分别指向基类对象（虚线部分表示丢失的派生部分）和派生类对象（左边表示基类部分，右边表示派生类部分）的基类指针（有箭头的矩形）和派生类指针（较宽的有两部分的矩形）。

每一部分中的垂直线表示四种成员函数。类型1是在基类中定义并在派生类中按原样继承

的函数。类型2是添加在派生类中而在基类中没有对应部分的函数。类型3是在基类中定义并在派生类中使用相同的函数名（以相同或者不同的标识）重定义了的函数。类型4是在基类中（作为虚函数）定义并在派生类中使用相同函数名和相同函数标识重定义的函数。

通过指针可以访问的函数用下划线表示。在图15-2a中，基类中定义的第3类和第4类函数被派生类中定义的函数隐藏。在图15-2b中，只能访问基类中定义的函数。在图15-2c中，只能访问基类中定义的函数，但在派生类中作为虚函数重定义的函数隐藏了基类中的相应函数，可被动态访问。在图15-2d中，只能访问在基类中定义的而且没有在派生类中重定义的方法。

表15-1总结了上述规则，其中列描述了对象的类型及指向这些对象的指针类型，行描述了不同种类的成员函数。

15.3.5 纯虚函数

基类的虚函数可能没有任何事情做，因为它在应用程序中没有任何意义。它们的工作只是为其派生类定义标准的接口。因此首先介绍虚函数。

例如，Person类中的write()方法没有包含任何代码，它从来没有被调用。在客户代码中，所有的write()方法调用（全局函数write()）都被解析为Faculty类的write()方法或Student类的write()方法。

事实上，Person类是一个没有任何任务的纯粹的泛化。在应用程序中没有Person对象，在全局函数read()中用new操作创建的对象或者属于Student类或者属于Faculty类。在本节开始描述问题时只是指出有两种记录：学生记录和教师记录。Person类首先作为一个抽象类被引入应用程序，它将教师对象和学生对象的某些共同特征合并为一个泛化类（见程序15-3）。然后，Person类用来定义派生类的继承层次（见程序15-4）。在程序的上一个版本中（见程序15-4），Person类用来定义虚函数write()的界面。

在实际生活中，Person类可能是非常有用的类，除了学校代号和名字外，还可包含出生日期、地址、电话号码及Faculty和Student对象的其他共同特点。除了数据外，Person类还可以定义许多方法，如改变名字、地址、电话号码，获取学校代号及Faculty和Student对象的其他共同数据。派生类可继承所有这些有用的函数。通过把这些（在Person类中定义的）消息发送给Faculty和Student类的对象，派生类的客户也可使用这些函数。我们并不是说Person类没有作用，而是说在这个应用程序中Person类的对象没有作用。该应用程序只需要从Person类派生的类对象。请注意两种说法之间的区别。

Person类的设计人员知道这个应用程序没有创建Person类的对象，也知道该类的对象无事可做。如果可以把这个事实通过代码而不是注释传达给客户端代码程序员（和维护人员）就好了。C++允许以这样的方式定义基类：试图创建这个类的对象将是不合法的，并会出现语法错误。

通过使用纯虚函数和抽象类可以实现这种定义。我们也不太肯定为什么两个术语“纯”和“抽象”用来表达同一个意思。纯虚函数是不能调用的虚函数（如Person类中的write()），如果程序试图调用这个函数将出现语法错误。抽象类是至少有一个纯虚函数的类（多于一个也是可以的）。创建抽象类的对象是不合法的。如果程序试图去动态创建或在栈中创建抽象类的对象，就会出现语法错误。

没有C++关键字表示纯虚函数和抽象类。相反，认为纯虚函数是一个成员函数，其声明“初始化”为0。这里，Person类的write()成员函数定义为纯虚函数。

```
struct Person {                          // abstract class
protected:
  char id[10];                           // data common to both types
  char* name;
public:
  Person(const char id[], const char nm[]);
  virtual void write() const = 0;        // pure virtual function
  ~Person();
} ;
```

当然，这里的赋值运算符并不表示赋值，只是在其他上下文中赋予一个符号额外的意义，这让人费解。如果添加另外一个关键字，例如pure或者abstract，就会是一个更好的设计。

纯虚函数没有实现。实际上，如果为纯虚函数提供实现（或激活纯虚函数）会出现语法错误。正是纯虚函数的存在使得类成为抽象（或部分）类。

除了不能直接创建对象实例外，抽象类还要求至少有一个派生类。如果派生类中实现了纯虚函数，那么这个派生类成为一个普通的类；如果派生类没有实现纯虚函数，那么这个派生类也成为一个抽象类。因此不能创建这个派生类的对象，这个派生类也应该至少有一个派生类。

派生类实现纯虚函数的方式与实现一般的虚函数的方式相同。这意味着派生类应使用与纯虚函数相同的名字、相同的标识及相同的返回类型。派生的模式应是public。下面的例子中，Faculty类实现了纯虚函数write()，Faculty类是一个一般类，而不是抽象类。

```
struct Faculty : public Person {         // regular class
private:
  char* rank;                            // for faculty only
public:
  Faculty(const char id[], const char nm[], const char r[]);
  void write () const;                   // regular virtual function
  ~Faculty();
} ;
```

事实上，这个派生类与程序15-5中的派生类相同。对于这个非抽象类，无法知道它是从抽象类派生而来还是从一般类派生而来。这完全不重要，因为对于Faculty类的用户而言，其基类Person是如何实现的并没有关系，只要客户代码不实例化这个抽象类的对象即可。

对于有虚函数的一般类，客户代码可以创建这个类的对象，给这些对象发送消息，如果需要还可以使用多态性。

不管怎么样，抽象类都是一个C++类，它可以包含数据成员和一般的非纯虚函数，包括虚函数在内。

如果一个类将纯虚函数仍作为纯虚函数继承而未对其函数体进行定义，这个派生类也是一个抽象类；不能创建这个类的对象。如果客户代码需要这个类的对象而不调用它的这些函数（因为这个函数还没有实现任何工作），可使用空的函数体。这时，这个类变成一般的非抽象类，可以创建这个类的对象。

```
class Base {                             // abstract class
```

```
public:
 virtual void member() = 0;          // pure virtual function
 .... } ;                            // the rest of Base class

class Derived : public Base {        // regular class
  public:
   void member()                     // virtual function
    { }                              // empty body: noop
 .... } ;                            // the rest of Derived class
```

这里，Base类是一个抽象类，不能创建这个类的对象。Derived类是一个一般类，可在栈中（作为命名了的变量）实例化它的对象，也可以在堆中（作为未命名的变量）实例化它的对象。Base类中的member()函数是个纯虚函数，不能调用它。Derived类中的member()函数是一个一般的虚函数，但调用它也没有操作。

```
Base *b;  Derived *d;          // Base and Derived pointers
b = new Base;                  // syntax error, abstract class
d = new Derived;               // OK, regular class, heap object
b = new Derived;               // OK, implicit pointer conversion
d->member();                   // OK, compile time binding, no op
b->member();                   // OK, run time binding
d->Base::member();             // linker error: no implementation
```

在派生类中用不同的标识对纯虚函数重定义，使得这个函数成为非虚函数。这里，Derived1类是从抽象类Base派生而来的，它没有重新定义无参数的纯虚函数member()，反而定义了一个有一个参数的member(int)函数。

```
class Derived1 : public Base {       // also an abstract class
  public:
   void member(int)                  // non-virtual function
    { }                              // empty body: noop
 .... } ;                            // the rest of Derived1 class
```

这意味着Derived1类也是一个抽象类，创建这个类的对象将是语法错误。由于这个类没有用作派生其他类的基类，它是一个没有用的类，因为没有办法使用它的任何函数。

```
class Derived2 : public Derived1 {   // regular class
  public:
   void member()                     // virtual function
    { }                              // empty body: noop
 .... } ;                            // the rest of Derived class
```

Derived2类从Derived1类派生而来，实现了虚函数member()，因而创建这个类的对象是合法的。类的对象可以使用动态绑定和静态绑定响应member()消息，但不能响应member(int)消息，因为Derived2类中定义的成员函数member()隐藏了这个函数。

```
Derived2 *d2 = new Derived2;       // OK, regular class, heap object
d2->member();                      // OK, static binding
b = new Derived2;                  // OK for virtual functions
b->member();                       // OK, dynamic binding
b->member(0);                      // syntax error
d2->member(0);                     // wrong number of parameters
```

注意，基类指针b指向派生类对象时，只可以激活以下函数：

• 在基类中定义的非纯虚函数（虚函数或非虚函数）。

• 在派生类中定义的虚函数。

它不能激活在派生类中定义的非虚成员函数，它是一个作用范围较小的指针。使用虚函数可以把它访问的范围扩展到所指向对象的派生部分。否则，它只能访问派生对象的基类部分。使用派生类指针可以访问派生类对象的基类部分和派生类部分。

15.3.6 虚函数：析构函数

当激活删除操作时，调用析构函数、撤销对象。调用哪一个析构函数呢？是指向对象的指针所属的类中定义的析构函数，还是由指针指向对象所属的类中定义的析构函数？

当指针和由指针指向的对象属于同一类时，答案很简单：就是那个类的析构函数。

```
Derived2 *d2 = new Derived2;        // OK, regular class, heap object
d2->member();                       // OK, static binding
b = new Derived2;                   // OK for virtual functions
b->member();                        // OK, dynamic binding
delete d2;                          // class Derived2 destructor
delete b;                           // ??
```

C++的析构函数是一般的非虚成员函数。当使用delete操作时，编译程序查找指针操作数的定义，查找指针所属类的定义，调用这个类的构造函数。所有这些发生在编译时，编译程序并不注意这个指针正指向对象的类。

当指针和对象属于同一个类时没有问题：分配给对象的动态内存（和其他资源）在析构函数代码被执行时还回。但当派生类指针指向基类对象时——当然我们不应该这样做，强大的派生类指针将要求弱小的基类对象做它不能做的事。

```
Person p;  Faculty f;                   // base and derived pointers
p = new Person("U12345678", "Smith");
f = p;                                  // syntax error: stay away
f = (Faculty*)p;                        // I insist I know what I do
delete f;                               // Faculty destructor
```

在这个例子中，Person对象激活Faculty类的析构函数，为这个对象中并不存在的rank数据成员调用delete操作时会出问题，其结果未定义。

当基类指针指向派生类对象时，调用基类析构函数。这可能会有问题也可能没有问题。如果动态内存在基类中而不是在派生类中处理，则不会有问题，基类析构函数将还回堆内存。如果是派生类处理堆内存，则基类析构函数不能还回堆内存，将导致内存漏洞。

```
Person *p;  Faculty* f;                 // base and derived pointers
f = new Faculty("U12345689", "Black", "Assistant Professor");
p = f;                                  // or p = (Person*) f;
delete p;                               // memory leak
```

这个例子中，delete操作激活了Person类的析构函数，这个析构函数将删除为name所分配的动态内存。由于没有调用Faculty类的析构函数，因而没有还回为rank分配的堆内存。

在程序15-5中，客户代码使用循环遍历基类指针数组，在程序开始执行时删除动态分配的每个对象。对于数据结构中的每个对象，将执行Person类的析构函数。

```
for (int j=0;  j < cnt;  j++)
{ delete data[j]; }                    // delete Person heap memory
```

就Faculty对象和Student对象而言，完全还回了其内存。无论对象是哪种类型，delete操作都一律删除这些对象。问题不在于对象的内存而是分配给派生类对象的堆内存，即图15-10中的最右边部分。Person类的析构函数删除了为name分配的堆内存，但没有删除为rank和major分配的堆内存。当通过基类指针将派生类对象撤销时，只调用了基类析构函数，而没有调用派生类的析构函数。

C++为此提供了一种办法：将基类的析构函数声明为虚函数。按惯例，这就让每个派生类的析构函数也变成虚函数。当基类指针使用delete操作时，目标类的析构函数以多态形式调用（如果有基类的析构函数，就再调用它）。

```
struct Person {                                    // abstract class
protected:
  char id[10];                                     // data common to both types
  char* name;
public:
  Person(const char id[], const char nm[]);
  virtual void write() const = 0;                  // pure virtual function
  virtual ~Person();                               // this makes the trick
} ;

struct Faculty : public Person {                   // regular class
private:
  char* rank;                                      // for faculty only
public:
  Faculty(const char id[], const char nm[], const char r[]);
  void write () const;                             // regular virtual function
  ~Faculty();                                      // now this is virtual, too
} ;
```

但这种方法有些不足。毕竟，我们首先应该记住的是：虚函数要求在基类和派生类中都使用相同的函数名。而这对析构函数就不适用了，因为每个析构函数与所属的类名相同，因而违反了虚函数规则。构造函数也类似，而且实际上C++中没有虚构造函数。

然而，从实际应用的角度考虑比逻辑上的完美更加重要。内存泄漏太危险，因此在C++中允许存在这种不一致，从而支持虚析构函数。

15.4　多继承：多个基类

在C++中，派生类可以有不止一个基类。在单继承中，类之间的继承层次是一棵树，基类在层次的最上面，派生类在基类的下方。

而在多继承中，类之间的继承层次可能是一个图而不是单继承的一棵树。这种类之间的关系比单继承中的关系复杂得多。与多继承相关的问题也比简单继承中的问题难于理解。

与单继承相似，多继承不是简化客户代码或使之易懂的技术，而是简化服务器代码的编写技术。与单继承不同的是，多继承允许服务器类的设计人员将多个类的特征混合在一个类中。

下面讨论一个简单例子。假设类B1为客户提供了public服务f1()，类B2为客户提供了public方法f2()。这几乎恰好是客户代码所需要的，但除了这两个服务外，客户代码还需要一个f3()方法。满足客户代码的一种可行办法是使用多继承，将B1和B2类的某些特征

合并到一个类中。

```
class B1
  { public:
       void f1();                    // public service f1()
    ... };                           // the rest of class B1

class B2
  { public:
       void f2();                    // public service f2()
    ... };                           // the rest of class B2
```

通过public继承B1和B2类，Derived类为其用户提供了每个基类所能提供的所有方法的集合，即方法f1()和f2()。这意味着为它的客户提供了所有三个服务（f1()，f2()和f3()），Derived类的设计者只需实现f3()函数就可以了。这的确很了不起。

```
class Derived : public B1, public B2          // two base classes
 { public:                                    // f1(), f2() are inherited
      void f3();                              // f3() is added to services
   ... };                                     // the rest of class Derived
```

现在，客户代码可以实例化Derived对象，并把消息发送给这些对象。不仅可以发送从两个基类中继承而来的消息，也可以发送在Derived类中增加的消息。

```
Derived d;            // instantiate Derived object
d.f1();  d.f2();      // inherited services (B1, B2)
d.f3();               // the service is added in the Derived class
```

我们可以看到派生类为其客户不仅提供了所有基类的数据和行为，还增加了自己的数据和行为。

开始时，C++没有多继承。但是C++的设计者Stroustrup抱怨说程序员“要求多继承”，这样现在的C++就有多继承了。不能肯定这有多少是迫于外在的压力，因为还有相当多的其他建议Stroustrup根本没有采纳。

多继承可以很好地自定制现有的类库，例如，为已存在的类增加或重定义成员。派生类表示的是多个基类的结合而不是单个基类的优化。每个父类都把自己的成员贡献给派生类，派生类是基类特点的联合。

使用多继承的例子很多，包括图形对象、NOW账户及在C++标准库中的iostream类等。

对于图形包，Shape和Position类作为基类产生Object类。Object类的对象结合了Shape和Position对象的特性。这是不太合理地使用多继承的例子。一个图形对象当然是一个形状，但是说一个图形对象也是一个位置就太勉强了。说一个图形对象有一个位置要更加自然。

对于NOW账户，它表示了存款账户和支票账户。这是使用多继承比较好的例子，因为NOW账户实际上结合了存款账户和支票账户的特性：它支付利息并允许写支票。但是如果我们和一个银行的官员进行讨论，他会说这基本上是正确的，但是仍然有一些异常情况使得NOW账户不同于存款账户和支票账户。这意味着，简单地将基本的特征合并在一起所得到的益处，往往被压缩那些不合适的特点所导致的缺点抵消。

对于C++的iostream类库，使用多继承将输入流类和输出流类的特征合并在一起，这是有意义的。得到的iostream类既支持输入操作也支持输出操作，而且在派生类中没有压缩

任何属性。

注意，C++并没有限制参与形成一个派生类的基类的数目。我们所举的例子中都只涉及到了两个基类。在下面的多继承的例子中，我们也会只使用两个基类。为什么不用三个或者四个呢？答案很简单，很难想出有意义的、不会让使用者迷糊的有三个或四个基类的多继承的例子。两个为什么比三个或者四个好呢？我们认为只有两个基类的多继承的例子同样也很复杂。

因此，建议大家要谨慎使用多继承。其优势不多，却很复杂。不使用多继承也常常足够支持客户代码。

15.4.1　多继承：访问规则

在多继承中，派生类继承了所有基类的所有数据成员和成员函数，派生类对象所占的内存空间是所有基类对象所占内存空间的总和（可能还要为了对齐加上额外空间）。

多继承的访问规则与单继承相同。派生类的方法可以无任何限制地访问所有基类的公共和受保护成员（数据和方法），但不能访问基类的私有成员。

继承模式也分为公共的、受保护的和私有的。在每种情况下，派生类都将继承所有基类的所有数据成员和成员函数，但访问权限将根据派生模式不同而不同。

多继承的派生模式与单继承相同。在公共派生时，每个基类成员，无论是私有、受保护还是公共的，在派生类的对象中拥有与在基类对象中一样的访问权限。这是一种最自然的继承模式。

在受保护派生时，受保护的和私有的基类成员在派生类中仍为受保护的和私有的，但公共的基类成员（数据和函数）在派生类中变成受保护的。由于派生类有访问受保护的基类成员的完整权限，受保护继承不会影响派生类的访问权限。与单继承相似，它会影响客户代码的访问权限。客户代码将失去使用公共基类成员的权限，派生类必须提供足够的服务给客户代码，而不能让客户代码访问公共的基类成员。

在私有继承时，所有基类成员在派生类中都变成私有的。与单继承相似，派生的缺省模式是私有派生。最好为每个基类分别定义派生模式。

下面考虑前面例子中的B1和B2两个基类。

```
class B1
  { public:
       void f1();                    // public service f1()
    ... };                           // the rest of class B1

class B2
  { public:
       void f2();                    // public service f2()
    ... };                           // the rest of class B2
```

我们将B1和B2的特征结合在派生类Derived类中，并在派生类中增加了另一个成员函数。

```
class Derived : public B1, B2        // two base classes
 { public:                           // f1(), f2() are inherited
      void f3();                     // f3() is added to services
   ... };                            // the rest of class Derived
```

这时，客户代码可以定义并使用Derived类的对象了。

```
Derived d;              // instantiate Derived object
d.f1();                 // inherited from B1
d.f2();                 // syntax error: f2() is private
d.f3();                 // the service is added in the Derived class
```

这里又再次说明，用于描述访问权限的public关键字和描述派生模式的public是不同的。描述访问权限时，public关键字的作用域包括所需的任何类成员，直到发现另一个访问权限关键字或到了类定义的末尾。描述派生模式时，public关键字的作用域只是一个标识符。

在上面的例子中，Derived类只公共继承了B1类；对于B2类则使用了缺省派生模式（私有继承）。因此，f2()方法在Derived类中变成私有的，因而不能被客户代码访问。

15.4.2 类之间的转换

多继承中的转换规则与单继承中的相似。如果公共地继承基类，派生类的对象可以隐式地转换为基类的对象，不需要显式进行这种转换。

这个规则的概念与单继承的相同。派生类的对象拥有基类对象的所有功能、数据和函数。只要派生模式不是公共的，将派生类对象转换为基类对象不会有任何功能的损失。

```
B1 b1;  B2 b2;  Derived d;
b1 = d;  b2 = d;            // OK: extra capabilities are discarded
d = b1;  d = b2;            // error: inconsistent state of object
```

将基类对象转换为派生类对象是不允许的。基类对象只拥有派生类对象中的部分数据和功能，而派生类中的其他数据和功能无处可找，因而这种转换是不安全的。

相似的规则也同样适用于指针和引用。基类的指针（引用）可安全地指向派生类的对象，派生类对象不仅可以处理基类指针所要求的工作，还能处理其他更多的工作。这是安全的。但基类指针只能激活派生类对象中的部分功能。

```
B1 *p1; B2 *p2;  Derived *d;
p1 = new Derived;  p2 = new Derived;      // OK: safe
d = new B1;  d = new B2;                  // syntax errors
d = p1;  d = p2;                          // syntax errors
d = (Derived*) p1;                        // OK: explicit cast
```

派生类的指针不能指向基类对象（上面例子的第三行）。基类对象缺乏派生类对象拥有的很多功能，也缺乏对派生类指针有效的成员。为了避免运行时错误，编译程序声明这种代码为语法错误。

类似地，基类指针（假设指向基类对象）不能拷贝给派生类指针（上面例子中的第四行）。这是不安全的，派生类指针可能要求基类对象所无法提供的服务，而编译程序无法捕获它。因此这种指针处理也被认为是语法错误。

那么，如果我们知道基类的指针的确指向一个派生类对象而不是基类对象，怎么办呢？与单继承一样，可以使用显式转换告诉编译程序我们知道自己在做什么（例子中的最后一行）。既然编译程序不会置疑我们，它就会直接接受我们的意见。因此，最好保证我们的想法是正确的。

同样的规则也适用于参数传递。如果函数的参数是指向基类对象的指针（或引用），可以使用派生类对象的地址作为参数调用这个函数，这是安全的。

```
void foo1 (B1 *b1)                    // derived objects have additional services
 { b1->f1(); }

void foo2 (B2 *b2)                    // derived objects have additional services
 { b2->f2(); }

void foo(Derived *d)                  // base objects cannot do that
 { d->f3(); }

B1 *b1 = new Derived;  B2 *b2 = new Derived;
Derived d;
foo1(&d);  foo2(&d);                  // both are OK: safe conversion
foo(b1);  foo(b2);                    // syntax errors: unsafe conversion
foo((Derived*)b1);  foo((Derived*)b2);     // pass at your own risk
```

在前一个例子中，foo1()和foo2()函数可以接收Derived类的对象作为实际参数，因为在这些函数内，参数只响应基类消息（f1()和f2()），而派生类对象是可以响应这些服务的。foo()函数不能接收基类指针，因为在这个函数内，参数要响应派生类的消息f3()，而基类对象不能访问它。另一方面，指向Derived类对象的b1和b2指针可以访问消息f3()，因此在以上代码的最后一行，使用显式转换将基类指针转换为Derived类指针。

在私有继承或受保护继承模式下，不允许将派生类对象隐式转换为基类对象。即使这种转换是“安全”的，客户代码也要使用显式转换（因为在这种情况下已不再“安全”）。对于任意模式的多继承，将基类对象转换为派生类对象时要使用显式转换。

15.4.3　多继承：构造函数和析构函数

派生类负责从基类继承而来的成员的状态。如同单继承一样，当创建一个派生类对象时要调用基类的构造函数。

给基类构造函数传递参数的机制与单继承类似：要使用成员初始化列表。在下面的例子中，基类B1有一个数据成员，基类B2有一个数据成员，派生类还有另外一个数据成员（动态分配的字符数组）。派生类Derived应提供带有三个参数的构造函数，以将数据传递给它的B1和B2部分的成员及它自己的数据成员。

```
class B1 {
  int m1;
public:
  B1(int);
  void f1(); ... };

class B2 {
  double m2;
public:
  B2(double);
  void f2(); ... };

class Derived:  public B1, public B2 {
  char* t;
public:
  Derived(const char*, double, int);
  ~Derived();
  void f3(); ... };
```

如果没有提供初始化列表，则调用基类的缺省构造函数。如果基类没有缺省的构造函数，就会出现语法错误。

在成员初始化列表中，Derived类构造函数用类名B1和B2（不是数据成员名）调用基类的构造函数，B1和B2出现在逗号分割开来的构造函数序列中。基类构造函数的参数通常来自于Derived类构造函数的参数列表（也可以是字面值）。

```
Derived::Derived(const char *s, double d, int i) : B1(i),B2(d)
 { if !(t = new char[strlen(s)+1]) == NULL)
     { cout << "\nOut of memory\n";  exit(1); }
   strcpy(t,s); }
```

所有的基类构造函数在调用派生类的构造函数之前调用，调用次序是按派生类定义中所列举的基类的次序，而不是按派生类构造函数的初始化列表中的次序。

与单继承类似，派生类的数据成员既可以在派生类构造函数体内初始化，也可以在成员初始化列表中初始化。

当派生类对象（动态或因为超出作用域而）撤销时，首先调用派生类的构造函数。

然后以与构造函数调用次序相反的顺序调用基类的析构函数。

15.4.4 多继承：二义性

多继承可能导致名字冲突，如果派生类和基类有同名的数据成员或成员函数，基类中的相应成员将被定义在派生类中的同名成员隐藏。

下面的例子中，Derived类有一个数据成员x与基类B1的数据成员同名，而且Derived类中还有一个成员函数f2()与基类B2中的成员函数同名。

```
class B1 {
protected:
  int x;                              // hidden by Derived::x
public:
  void f1();  ... };

class B2 {
public:
  void f2();  ... };                  // hidden by Derived::f2()

class Derived:  public B1, public B2 {
protected:
  float x;                            // hides B1::x
public:
  void f2();                          // it hides B2::f2()
  void f3()
   { x = 0; }  ... };                 // Derived::x is used
```

在这个例子中，Derived类的对象有两个数据成员x，从B1继承而来的数据成员x被Derived类中定义的x隐藏了，从B2继承而来的成员函数f2()被Derived类中添加的f2()隐藏了。

客户代码和Derived类代码可以使用显式的作用域运算符来忽略作用域规则。

```
void Derived::f3()
 { B1::x = 0; }                       // disregard Derived::x
```

```
Derived d;
d.f2();                                  // Der::f2();
d.B2::f2();                              // B2::f2();
```

派生类和基类之间的名字冲突不是很常见。通常，派生类的设计者可浏览基类的设计，在认为有害时可避免冲突。

基类之间的成员名字也可能冲突。这更经常发生而且更难以处理，因为基类通常是分别开发的，很难互相协调以避免名字冲突。

如果两个基类有同名的数据成员或成员函数，在派生类对象中对它们都会有副本。语言没有为数据和函数的访问提供预定义的优先级规则。实际上我们使用显式的限制来解决二义性问题，即在客户代码和派生类中都使用作用域运算符。

在下面的例子中，两个基类都有public成员函数名为f1()。这意味着除非客户代码明确指出使用哪个基类中的版本，否则不能使用任一版本的函数。

```
class B1 {
public:
  void f1(); ... } ;

class B2 {
public:
  void f1(); ... } ;

class Derived : public B1, public B2 {
public:
  void f3(); ... } ;

Derived d;
d.f1();                                          // ambiguous message
d.B1::f1();  d.B2::f1();  d.f3();                // OK
```

这种消除二义性的办法违反了面向对象程序设计的原则。这种解决办法将任务加到客户代码中而不是服务器类中。如果发现了这样的设计一定要避免它。

另一更好的办法是让Derived类将客户代码和成员函数名字的二义性隔离开来。

```
class Derived:  public B1, public B2 {
public:
  void f1() { B1::f1(); }                        // one-liners
  void f2() { B2::f1(); }
  void f3();    ... } ;

Derived d;
d.f1();  d.f2();  d.f3();                        // client is insulated
```

这个办法要好得多。是服务器代码（Derived类）处理了二义性问题，而客户代码与这个问题隔离开了。毕竟，当客户代码使用Derived类作为它的服务器类时，它不必了解服务器类的设计细节，如：它是继承而来的，它从哪些类继承而来，它必须处理哪些冲突等细节问题。客户代码只需要知道如何调用f1()、f2()的f3 ()以完成任务就可以了。

如果两个（或更多）基类有同名的数据成员（相同或不同类型），在派生类对象中也会有相应的副本，因此也会产生二义性。

```
class B1 {
```

```
protected:
  int m;
public:
  B1(int);
  void f(); ... };

class B2 {
protected:
   double m;
public:
   B2(double);
   void f(); ... };

class Derived : public B1, public B2 {
  char* t;
public:
  Derived (char*,double,int);
  void f3() { cout << "m=" << m << endl; }       // ambiguity
  ... };
```

数据成员之间的名字冲突也应该由派生类来解决，以避免二义性，并保护客户代码。这样，必须使用作用域运算符。

```
void Derived::f3()
{ cout << "m=" << B1::m << endl; }              // no ambiguity
```

15.4.5 多继承：有向图

当不止一次地继承一个基类时，会出现最难处理的二义性。通常在C++中不允许这样做，一个类在某个派生类的派生列表中只能显式地出现一次。

```
class B {
public:
   int m;  . . . . } ;

class Derived:  public B, public B          // syntax error
 {  . . . .  } ;
```

这样做没有什么意义，而且属于语法错误。但同一个类在继承层次中可出现多次。不同的基类可以有共同的父类；这样，一个父类在派生序列中不止出现一次，其数据在派生类中有多个副本。

```
class B1 : public B  {                      // class B is above
  protected:
     int mem;
  public:
     void f1(); ... };

class B2 : public B  {                      // class B is above
  protected:
     int mem;
  public:
     void f2(); ... };

class Derived : public B1, public B2 {      // inherited from B twice
  public:
```

```
    void f3();  ... };
```

在这个设计中，Derived类有两个名为mem的数据成员，分别从不同基类继承而来。它们虽有相同的名字，但指向不同的内存地址。它们在程序中的角色也不同：来自不同的类。处理这种情况非常令人头疼，而且这个难题不应该推给客户去解决。

下面例子中的数据成员m更难处理。Derived类的每个对象都有这个数据成员的两个实例：一个从B1继承而来，另一个从B2继承而来。为相同的基类数据成员的多个实例分配各自的内存空间是个浪费。而且，这两个数据成员的功能相同——来自相同的类，只是一个为Derived类的B1部分服务，另一个为Derived类的B2部分服务。

C++中有一个修正此问题的有趣做法。它让程序员有机会显式声明使用同一数据和函数的两个（或者多个）副本是没有必要的。如果这是缺省情况就好了，可以让那些喜欢麻烦的人有权力提出要求：使用相同数据和函数的两个副本是必要的。

方法就是定义基类为虚基类。在以后会在多继承中使用的派生类的声明中使用virtual关键字即可。

```
class B  {                                      // common base class
  int m;
public:
  void f();  ...    } ;

class B1 : virtual public B  {                  // virtual base
  protected:
     int mem;
  public:
     void f1(); ... };

class B2 : virtual public B  {                  // virtual base
  protected:
     int mem;
  public:
     void f2(); ... };

class Derived : public B1, public B2 {          // works by magic
  public:
    void f3();  ... };
```

现在，Derived类只有从B类继承而来的数据和函数的一个副本了。注意，为了解决Derived类的问题，必须由它的基类B1和B2的设计人员来定义这些类为虚类。这也违背了一个原则：基类不知道其派生类的情况，而派生类知道其基类。

最后还要说明：不要将这个上下文环境中使用的virtual关键字与虚函数中的virtual关键字相混淆，它们完全不同。如果有两个不同的关键字会好一些。如果没有多继承可能会更好一些。

15.4.6　多继承：有用吗

我们不确定能否公正地回答多继承是否有用，但是我们认为，使用多继承的设计复杂性（在这里讨论的只是其中的一部分）要超过其优越性。

那么，如果我们不得不为客户设计一个类，它包含几个类所提供的服务集合，怎么办

呢？我们的答案是：使用复合或有继承的复合。

我们再来讨论本节开始时的第一个例子。该设计的目的是让客户代码调用函数f1()、f2()和f3()。函数f1()和f2()在B1和B2中已实现，还需要实现函数f3()。下面是实现f1()和f2()的两个类。

```
class B1
  { public:
       void f1();                    // public service f1()
    ... };                           // the rest of class B1

class B2
  { public:
       void f2();                    // public service f2()
    ... };                           // the rest of class B2
```

当使用多继承时，在新类Derived中实现所需的函数f3()。

```
class Derived : public B1, public B2    // two base classes
 { public:                              // f1(), f2() are inherited
      void f3();                        // f3() is added to services
   ... };                               // the rest of class Derived
```

我们可以不这样做，而是创建一个Derived类，从B1类中继承f1()函数。为了向Derived的客户提供函数f2()，可以将B2类作为Derived类的一个成员域。

```
class Derived : public B1 {          // single inheritance
   B2 b2;                            // class composition
public:
   void f2() { b2.f2(); }            // one-liner
   void f3(); ... };
```

现在，客户代码可以实例化Derived类的对象，并可以像多继承中那样把消息发送给这些对象。

```
Derived d;              // instantiate Derived object
d.f1();                 // inherited services (B1)
d.f2();                 // passed from B2 through Derived
d.f3();                 // added in the Derived class
```

当然，客户代码不用知道其服务器类Derived的设计细节。Derived类提供了所要求的服务，那就足够了，而且没有多继承所带来的复杂性。

15.5 小结

本章，我们讨论了继承的高级应用。其核心是基类和派生类有相同的功能。这样，至少在某些时候可以用一个类的对象代替另一个类的对象使用。

在需要基类对象时，使用派生类的对象总是安全的。这种转换是安全的，但是意义不大。要求派生类对象只会处理基类对象所能处理的工作，但是它实际上可以处理更多的事。

然而，虽然对于指针（与引用）也同样如此，情况却有趣得多。这意味着在需要使用派生类指针时可以使用基类指针，也就是说，基类指针可以指向派生类的对象。

为什么我们要这样做呢？这样做的最通用上下文环境是可以处理不同类的对象集合。

这一直是程序设计语言的一个问题。现代语言支持的所有集合都是同构的。C++数组不能

包含不同类的成员，C++链表也不能使用不同类型的结点，只有使用继承才能允许处理不同类的对象。而这些类并非完全不同。本章讨论的异构集合也不能包含任意类的对象，这些对象的类必须是有继承关系的。

当处理（通过继承相关的）异构对象集合时，有四种消息可传送给集合中的对象：

- 集合中的每个对象都能响应的消息，即那些在继承层次的基类中定义的，而且在派生类没有重写的方法。
- 集合中某些种类的对象可响应的消息，即那些在派生类中定义的，而且在基类中没有相同名字的方法。
- 集合中所有种类的对象都可响应的消息，但是这些消息在基类和派生类中作为非虚函数定义（有相同或不同接口）。
- 集合中每种对象都可响应的消息，这些消息在基类和派生类中使用相同接口定义为虚函数。

第一种类型的消息可使用基类指针访问。在访问集合中的对象时不需要转换。

第二种类型的消息只能使用派生类指针传送。当从集合中取对象时，必须将基类指针转换为对象所属类的指针，只有这样才能访问第二种消息。这种转换是不安全的，我们应该知道自己在做什么（即对象可以响应这个消息），因为编译程序不能保护我们。

如果对象要访问在派生类中定义的消息，第三种消息也需要转换。基类中的消息被派生类中的相应消息隐藏了。

第四种消息不需要转换。即使这些消息使用的是基类指针，在运行时系统会根据指针指向的对象类型来解释（动态绑定）。使用虚函数的设计，可以将不同种类对象的不同处理算法封装到同名函数中。

使用虚函数会浪费一些内存空间，对程序性能也会有一定影响。使用了虚函数的类的每个对象都有一个隐藏了的数据成员（该成员描述对象的类型），或有一个指向有效的虚函数地址表的指针。每当调用虚函数时，用这个指针查找需要的目标代码，这个额外的间接步骤将增加执行时间。但这种浪费不是很大，对于大多数应用程序而言不是问题。

我们还讨论了多继承，我们认为其复杂性不亚于其他语言特征。同时，多继承的实用范围较小，所有使用多继承处理的事都可以将继承和复合混用来解决。不要使用或者尽量少使用多继承。

虚函数就完全不一样了，它们在C++程序设计中十分流行。它们很常用，所以也许反对使用它在策略上是不正确的。但是请记住虚函数机制是脆弱的。一定要用公共继承，而且在继承层次中的每个类中都使用相同的函数名、参数列表、返回类型，甚至const等修饰符。如果有一点不一致，程序都会调用一个完全不同的函数，这个函数只是有相同的名字而已，而我们可能没有注意到。

强调一点，在任何可以合理地用同一函数名描述不同的相关对象的操作情况下，使用虚函数。

第16章 运算符重载的高级应用

在第10章（数值类）和第11章（非数值类），我们已对C++重载运算符进行了讨论。本章，我们将对这些内容进行扩展，介绍C++中更多重载运算符的应用。对有些人来说，任何运算符函数的重载都是很奇怪的，在他们看来，本章的内容可能与大多数C++程序员的日常工作相距甚远。

这可能是正确的，也许不必经常编写高级的重载运算符，但这些论题并没有脱离许多C++程序员每天用到的知识。高级运算符是任何标准或非标准的C++库的重要组成部分，理解它们是如何工作的将很有用。当然，在学习C++时这些内容并不是首要的。但是，这些运算符对智力的确是很大的挑战，从美学角度来讲也是很让人兴奋的。花时间和精力学习它们一定会对我们大有帮助。

16.1 运算符重载简介

重载运算符是程序员定义的函数，它有一个特别的名字（由关键字operator和运算符符号或其他符号组成）。重载运算符也称为运算符函数、重载运算符函数或就是运算符。它们为操纵程序员定义类的对象提供了很方便的运算符语法。

在C++中，为了将程序员定义类型的对象与内部数据类型的变量以同样方式对待，特提供了重载运算符。如果我们可以将两个数据值相加，没有理由不能把两个Account对象相加。如果我们可以将三个数据值相加，也应该可以将三个Account对象相加，等等。重载的运算符可以让我们这样做。

C++语言为内部数据类型定义基本运算符的意义。而重载运算符的意义是程序员而不是语言本身定义的。这些意义不能随意定义，应该根据被加、乘或进行其他运算的对象的性质来定义。但是程序员可以很自由地定义重载运算符的意义，因此很容易滥用，导致重载运算符的意义不直观不清晰。这种滥用的一个典型例子就是，在第10章中我们为显示一个Complex数字的域而添加的那个二元运算符（见程序10-4）。在客户代码中使用这个运算符会让任何维护人员费解。例如，很少有人可以正确猜测到+x是表示以特定的格式显示对象x的域。

我们可以自由地选择重载运算符的内容，但在选择重载运算符的名字时要受到一些限制。运算符函数的名字应该包括关键字operator，后接C++中的任意合法运算符（允许如==或+=两个符号的运算符）。

以上规则有五个例外情况：即不能是“.”、“.*”、“::”、“?:”或“sizeof”运算符。重载运算符可以定义为类的成员函数（因此可以作为消息），也可以作为最高层的全局函数（这个函数通常是作为重载运算符的操作数对象的友元）。如果重载运算符作为类的成员函数，可以有任何合适的参数。

消息的目标对象将作为第一个操作数。如果重载运算符作为全局函数，则它至少有一个类类型的参数，而不能只有内部数据类型的参数。这些限制对内存管理的运算符（new、delete、delete[]）不起作用。

基类中的重载运算符可由派生类继承。显然，这些运算符不能访问在派生类中定义的成员，因为派生类成员在基类的作用域之外，因而不能用基类的方法进行访问。重载的赋值运算符是个例外，它不能由派生类继承，或者倒不如说，它只能访问派生类对象中与基类相关的部分，而不能访问派生类对象中的与派生类相关的部分。这样，继承层次中的每个类如果需要，都要分别定义它自己的赋值运算符。

重载运算符中的运算优先级与内部数据类型中的运算符优先级相同。例如，无论在程序员定义的类中意义如何，乘法运算都比加法运算的优先级要高。重载运算符的表达式语法也与内部数据类型中的表达式语法相同。例如，无论是基本的还是重载的，二元运算符都一直出现在它的两个参数之间。（但是在本章中会有一些例外。）

重载运算符的操作数个数也与内部数据类型中的相应运算符的操作数个数相同。二元运算符重载后依然是二元运算，要求有两个操作数。作为全局非成员函数时（例如友元），重载的二元运算符必须有两个参数。作为类成员函数时，重载的二元运算符只需要显式给出一个参数，因为另一个参数是消息的目标对象。

类似地，内部数据类型中的一元运算符重载后依然是一元的。如果将一元重载运算符作为全局的非成员一元运算符（如友元），它只有一个参数。如果将该重载运算符作为成员函数定义（作为消息发送给目标对象），它将没有参数。

下面考虑一个简单例子，其中Account类与第13章所讨论的类似。该类维护关于拥有者姓名及当前收支情况的信息，并允许客户代码访问对象数据成员的值以及执行存款和取款操作。

除了有四个内联成员函数外，该类还有一个一般的构造函数。它不需要缺省的构造函数，因为只有在需要时才在堆上创建Account对象。只有当事先创建类的对象而不知道拥有者的姓名和初始收支情况时，使用缺省构造函数才可能有用。

由于该类动态地管理内存，为它增加拷贝构造函数和赋值运算符可能比较好，或者定义这些成员函数原型为private（详见第11章）。为了简单起见，我们在这里不这样做，因为我们不按值传递Account对象，不用一个Account对象的数据初始化另一个Account对象，也不会将一个Account对象赋值给另一个Account对象。在实际生活中，防止无意地滥用Account对象是非常重要的。

```
class Account {                                          // base class of hierarchy
protected:
  double balance;                                        // protected data
  char *owner;
public:
  Account(const char* name, double initBalance)          // general
  { owner = new char[strlen(name)+1];                    // allocate heap space
    if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
    strcpy(owner, name);                                 // initialize data fields
    balance = initBalance; }
  double getBal() const                                  // common for both accounts
  { return balance; }
  const char* getOwner() const                           // protect data from changes
  { return owner; }
  void withdraw(double amount)
  { balance -= amount; }                                 // pop responsibility up
  void deposit(double amount)
  { balance += amount; } } ;                             // increment unconditionally
```

我们要创建一个Account指针数组，动态地创建Account对象，初始化它们，查找属于某给定拥有者的账户，并进行存款和取款。同样出于简单起见，我们使用的是程序中给定的数据，而不是从外部文件读入的数据。

程序16-1是该例子的源代码。createAccount()函数动态地创建了一个Account对象，用两个参数调用Account构造函数，并返回指向新分配的对象的指针。processRequest()函数建立了ios标志，要求按固定的格式打印浮点数及其尾部的0，并在对象内查找顾客姓名，如果没有找到则打印一条消息；否则该函数提示用户输入交易代码，要求输入交易的金额，并对交易（存款或取款）进行处理。

程序16-1 用程序员命名的方法处理Account类的例子

```
#include <iostream>
using namespace std;

class Account {                                          // base class of hierarchy
protected:
  double balance;                                        // protected data
  char *owner;
public:

  Account(const char* name, double initBalance)          // general
  { owner = new char[strlen(name)+1];                    // allocate heap space
    if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
    strcpy(owner, name);                                 // initialize data fields
    balance = initBalance; }

  double getBal() const                                  // common for both accounts
  { return balance; }

  const char* getOwner() const                           // protect data from changes
  { return owner; }

  void withdraw(double amount)
  { balance -= amount; }                                 // pull responsibility up

  void deposit(double amount)
  { balance += amount; }                                 // increment unconditionally
};

Account* createAccount(const char* name, double bal)
{ Account* a = new Account(name,bal);                    // account on the heap
  if (a == 0) { cout << "\nOut of memory\n";  exit(0); }
  return a; }

void processRequest(Account* a[], const char name[])
{ int i;  int choice;  double amount;
  cout.setf(ios::fixed,ios::floatfield);
  cout.precision(2);
  for (i=0;  a[i] != 0;  i++)
  { if (strcmp(a[i]->getOwner(),name)==0)                // search for name
    { cout << "Account balance: " << a[i]->getBal() << endl;
      cout <<"Enter 1 to deposit, 2 to withdraw, 3 to cancel: ";
      cin >> choice;                                     // transaction type
      if (choice != 1 && choice != 2) break;             // get out
      cout << "Enter amount: ";
```

```
      cin >> amount;                                          // transaction amount
      switch (choice) {                                       // select further path
        case 1: a[i]->deposit(amount);                            // unconditional
                break;
        case 2: if (amount <= a[i]->getBal())                     // enough funds?
                  a[i]->withdraw(amount);
                else
                  cout << "Insufficient funds\n";
                  break; }                                        // end of switch scope
      cout << "New balance: "<< a[i]->getBal() << endl;         // OK
      break; } }                                              // end of search loop
    if (a[i] == 0)
       { cout << "Customer is not found\n"; } }

int main()
{
  Account* accounts[100];  char name[80];                     // program data
  accounts[0] = createAccount("Jones",5000);                  // create objects
  accounts[1] = createAccount("Smith",3000);
  accounts[2] = createAccount("Green",1000);
  accounts[3] = createAccount("Brown",1000);
  accounts[4] = 0;
  while (true) // process requests
  { cout << "Enter customer name ('exit' to exit): ";
    cin >> name;                                              // accept name
    if (strcmp(name,"exit")==0) break;                        // test for end
    processRequest(accounts, name);                           // next transaction
       }
  return 0;
  }
```

main()函数定义了一个Account的指针数组，并调用createAccount()创建Account对象。在循环中，提示用户输入顾客姓名，并调用processRequest()处理交易。程序运行的一个实例如图16-1所示。

```
Enter customer name ('exit' to exit): Brown
Account balance: 1000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 2
Enter amount: 2000
Insufficient funds
New balance: 1000.00
Enter customer name ('exit' to exit): Brown
Account balance: 1000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 2
Enter amount: 500
New balance: 500.00
Enter customer name ('exit' to exit): Smith
Account balance: 3000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 1
Enter amount: 2000
New balance: 5000.00
Enter customer name ('exit' to exit): Simons
Customer is not found
Enter customer name ('exit' to exit): exit
```

图16-1　程序16-1的输出结果

在这个例子中，Account类依赖于其客户代码检查取款交易是否合法。该方法的优点是Account成员函数与用户界面无关，它们只负责访问Account数据成员。其不足是数据还需上推到客户代码作进一步处理，而不是将职责推向服务器代码处理。我们选用该方案的主要理由是便于使用重载的运算符。

首先要变成重载运算符的候选者是Account的成员函数deposit()和withdraw()。将它们转换为运算符函数所要做的是去掉当前的函数名（deposit和withdraw），并用新的名字来代替（operator+=和operator-=），不需要其他改变。

```
void operator -= (double amount)
{ balance -= amount; }              // client tests feasibility

void operator += (double amount)
{ balance += amount; }              // increment unconditionally
```

客户函数processRequest()不再调用deposit()和withdraw()成员函数，而是使用表达式语法：在左操作数（消息的目标）和右操作数（消息的参数）之间插入运算符。

```
switch (choice) {
  case 1: *a[i] += amount;          // a[i]->deposit(amount);
          break;
  case 2: if (amount <= a[i]->getBal())
              *a[i] -= amount;      // a[i]->withdraw(amount);
          else
              cout << "Insufficient funds\n";
          break; }
```

注意，消息的目标是一个Account指针。因此，该指针在表达式中使用时必须对它进行间接引用。这样不是很方便，但不要紧。至少没有决定应该使用点选择符（当消息的目标是一个对象或者一个引用时）还是箭头选择符（当目标是指针时）重要。

当然，这些表达式语法的真正意义是一个函数调用，将消息发送给表达式a[i]->operator+=(amount)和a[i]->operator-=(amount)左边的操作数。

程序16-2中使用了重载运算符函数来代替程序员自命名的方法，与程序16-1大致相似。在开始处理的交互阶段之前，main()函数调用函数printList()来遍历Account指针列表，并打印由这些指针所指对象的内容（见图16-2）。注意语句对输出内容格式化，使姓名左对齐，账户收支情况右对齐。

程序16-2 用重载运算符方法处理Account类的例子

```
#include <iostream>
using namespace std;

class Account {                                     // base class of hierarchy
protected:
  double balance;                                   // protected data
  char *owner;
public:

  Account(const char* name, double initBalance)     // general
  { owner = new char[strlen(name)+1];               // allocate heap space
    if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
```

```
        strcpy(owner, name);                              // initialize data fields
        balance = initBalance; }

  double getBal() const                                   // common for both accounts
  { return balance; }

  const char* getOwner() const                            // protect data from changes
  { return owner; }

  void operator -= (double amount)
  { balance -= amount; }                                  // pull responsibility up

  void operator += (double amount)
  { balance += amount; }                                  // increment unconditionally
} ;
Account* createAccount(const char* name, double bal)
{ Account* a = new Account(name,bal);                     // account on the heap
  if (a == 0) { cout << "\nOut of memory\n";  exit(0); }
  return a; }

void processRequest(Account* a[], const char name[])
{ int i;  int choice;  double amount;
  cout.setf(ios::fixed,ios::floatfield);
  cout.precision(2);
  for (i=0;  a[i] != 0;  i+-)
  { if (strcmp(a[i]->getOwner(),name)==0)                 // search for name
     { cout << "Account balance: " << a[i]->getBal() << endl;
       cout <<"Enter 1 to deposit, 2 to withdraw, 3 to cancel: ";
       cin >> choice;                                     // transaction type
       if (choice != 1 && choice != 2) break;
          cout << "Enter amount: ";
          cin >> amount;                                  // transaction amount
          switch (choice) {
            case 1: *a[i] += amount;                      // a[i]->operator+=(amount);
                    break;
            case 2: if (amount <= a[i]->getBal())
                      *a[i] -= amount;                    // a[i]->operator-=(amount);
                    else
                      cout << "Insufficient funds\n";
                    break; }                              // end of switch scope
          cout << "New balance: "<< a[i]->getBal() << endl;
       break; } }                                         // end of search loop
    if (a[i] == 0)
      { cout << "Customer is not found\n"; } }

void printList (Account* a[])
{ cout << "Customer List:\n\n";
  for (int i=0;  a[i] != 0;  i++)
  { cout.setf(ios::left, ios::adjustfield); cout.width(30);
    cout << a[i]->getOwner();
    cout.setf(ios::right, ios::adjustfield); cout.width(10);
    cout << a[i]->getBal() << endl; }
  cout << endl; }

int main()
{
  Account* accounts[100];  char name[80];                 // program data
  accounts[0] = createAccount("Jones",5000);              // create objects
```

```
  accounts[1] = createAccount("Smith",3000);
  accounts[2] = createAccount("Green",1000);
  accounts[3] = createAccount("Brown",1000);
  accounts[4] = 0;
  printList(accounts);
  while (true)                                        // process requests
  { cout << "Enter customer name ('exit' to exit): ";
   cin >> name;                                       // accept name
      if (strcmp(name,"exit")==0) break;              // test for end
      processRequest(accounts, name);                 // next transaction
      }
return 0;
}
```

```
Customer List:

Jones                                   5000.00
Smith                                   3000.00
Green                                   1000.00
Brown                                   1000.00

Enter customer name ('exit' to exit): Smith
Account balance: 3000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 1
Enter amount: 1000
New balance: 4000.00
Enter customer name ('exit' to exit): exit
```

图16-2 程序16-2的输出结果

与processRequest()相似，printList()函数在数组中找到空指针（这个指针起岗哨值的作用）之前迭代列表。注意这两个函数中循环头的区别：在printList()中，下标i是循环中的局部变量；在processRequest()中，下标对于该循环是全局变量（但局部于该函数作用域）。之所以有区别是因为，在printList()中循环结束后不再需要下标值，迭代总是从列表开始到最后；而在processRequest()中，迭代有可能未到达列表结束时就停止了（如果找到了该名字），因此processRequest()需要了解这些。

将重载运算符作为全局函数来实现很简单：消息的目标对象作为函数的第一个参数。运算符将使用第一个参数的数据成员，而不是使用目标对象的数据成员。下面是作为全局函数实现的两个运算符。

```
void operator -= (Account &a, double amount)        // global function
  { a.balance -= amount; }                          // pop responsibility up

void operator += (Account &a, double amount)
  { a.balance += amount; }                          // increment unconditionally
```

由于这两个函数要访问Account类的非公共成员，因而要将它们声明为Account类的友元。有些程序员认为这样做是自找麻烦，因为它需要额外的工作。但是，正如我们在前面提到的，现代程序设计方法并不认为编写额外的代码是缺点，如果它能产生更加容易理解的代码的话。我们只用写一次额外的声明，但是我们（和其他人）需要在程序开发、测试和维护过程中读许多次代码。

在本例中，在类中添加对友元函数的声明，可以清晰地表示这些函数属于这个类。它们物理上属于这个类，也就是说，只有Account类的对象才能使用它们。这些函数在概念上也属于这个类，即它们是该类所提供操作的一部分。友元函数的语法不同于成员函数的语法，但是技术上只有很小的不同。使用友元函数常常会破坏数据的封装性，并使程序各部分之间出现多余的依赖关系，但这些对于重载运算符函数都不是问题。

```
class Account {                                          // base class of hierarchy
protected:
       double balance;                                   // protected data
       char *owner;
public:
  Account(const char* name, double initBalance)          // general
  { owner = new char[strlen(name)+1];                    // allocate heap space
    if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
    strcpy(owner, name);                                 // initialize data fields
    balance = initBalance; }
  double getBal() const                                  // common for both accounts
  { return balance; }
  const char* getOwner() const                           // protect data from changes
  { return owner; }
friend void operator-= (Account &a, double amount);      // operators
friend void operator+= (Account &a, double amount);
} ;
```

从成员函数运算符转换成友元运算符并不改变客户代码中的表达式语法。

```
switch (choice) {
  case 1: *a[i] += amount;                  // operator+=(*a[i],amount);
          break;
  case 2: if (amount <= a[i]->getBal())
            *a[i] -= amount;                // operator-=(*a[i],amount);
          else
            cout << "Insufficient funds\n";
          break; }                          // end of switch scope
```

但这些代码的意义发生了变化。表达式语法仍然表示一个函数调用，但调用的是全局函数。在函数调用中不再需要目标对象。相反，参与运算的对象作为实际参数传递给函数。编译程序将以下面的形式来理解上面的客户代码。

```
switch (choice) {
  case 1: operator+=(*a[i],amount);        // a.k.a. *a[i]+=amount;
          break;
  case 2: if (amount <= a[i]->getBal())
              operator-=(*a[i],amount);    // a.k.a. *a[i]-=amount;
          else
              cout << "Insufficient funds\n";
          break; }                         // end of switch scope
```

注意，实际参数必须进行间接引用，因为a[i]是指向Account对象的指针，而不是对象本身。该引用参数必须用对象的值而不是指针的值来初始化。因此，该函数调用需要进行间接引用。

重载运算符为我们编写客户代码提供了简洁的表达方式，但它不能解决软件工程中的本质问题。用重载运算符处理的事情都可由常规的成员函数来处理，如程序16-1所表明的那样。

16.2 一元运算符

一元运算符只有一个操作数。这些运算符包括加1和减1运算符、求反运算符（negation operators）、逻辑和按位非运算符、正号负号运算符、转换运算符、地址运算符和间接引用运算符，以及new、delete、sizeof运算符等。所有这些运算符（除了sizeof之外）都可以用作重载运算符。

当然，并不是每个运算符对于每个类都有它特殊的意义。在第10章中，我们将正号运算符重载为Complex类的输出运算符，这样容易让人费解。因此，重载一元运算符不是很常见。但在某些情况下，使用重载的一元运算符可以帮助直观地理解客户代码。本节我们将讨论几个重载的一元运算符的例子。

16.2.1 ++和--运算符

在C++中，经常使用加1和减1运算符，尤其在处理文本时，将指针加1（或减1）可以与访问当前字符联合起来使用。

```
void printString(const char data[])      // text does not change
{ const char *p = data;                  // point to start of data
  while (*p != 0)                        // go until the end of data
  { cout << *p;                          // print current character
    ++p; }                               // point to next character
  cout << endl; }
```

在这个例子中，将字符数组（作为常量）传递给全局函数，并依次显示每个字符。指针p首先指向data[]数组的第一个字符，然后每次加1直到遇到终止符0。尽管看上去像是使内存地址加1的很低层次的控制，但实际上它是非常抽象的，因为该操作并没有显示存储管理的真实细节——如地址改变了多少，是实际增加了还是减少了等。

例如，无法保证一个参数数组在内存中是从低地址到高地址放置的。在本书的实例中，物理地址沿数组结束的方向减小。因此，当将指针加1时，不能保证指针的实际内容增加。它所表示的是设置该指针来访问下一个数组成员，无论该成员的大小是多少（可能比1大），也不管指针移动的方向是向上还是向下。

但是通过客户代码来“公开地”访问数组元素容易出错，访问数组范围之外的单元在编译时不会认为是语法错误。这种访问可能会在运行时使程序崩溃，程序可以无警告地产生不正确的结果，或者没有任何反应地产生正确的结果，直到内存的使用发生改变，从而导致灾难性后果。

将数据和操作结合在一个类中，可以防止客户端代码程序员不正确地访问数据，也可以给客户端代码程序员提供处理对象的工具以防止错误。下面例子中的String类与第11章中所讨论的大致相似。

```
class String {
  int size;                              // string size
  char *str;                             // start of internal string
  void set(const char* s);               // private string allocation
public:
  String (const char* s = "")            // default and conversion
    { set(s); }
  String (const String &s)               // copy constructor
```

```
    { set(s.str); }
  ~String()                                  // destructor
  { delete [ ] str; }
  String& operator = (const String& s);      // assignment
 int getSize() const;                        // current string length
 char* reset() const; } ;                    // reset to start of string
```

该类的数据成员str指向一个动态分配的数组，该数组的大小存储在数据成员size中。类的构造函数和赋值运算符使用私有成员函数set()。该函数接收一个字符数组（标注为const）作为参数，动态分配堆内存，将数据成员str指针指向新分配的内存，并用参数所提供的字符数组初始化动态分配的内存。

```
void String::set(const char* s)
 { size = strlen(s);                  // evaluate size
   str = new char[size + 1];          // request heap memory
   if (str == 0) { cout << "Out of memory\n"; exit(0); }
   strcpy(str,s); }                   // copy client data to heap
```

这是动态内存管理的一个典型设计。该私有函数提供了很大的方便性，因为它将构造函数和赋值运算符所共同使用的操作封装起来。

为了动态管理内存，String类提供了一个转换构造函数（也可作为缺省构造函数，因为有缺省参数值）、一个拷贝构造函数、一个赋值运算符和一个析构函数。缺省构造函数将一个空字符串传递给set()。转换构造函数将它自己的字符数组参数传递给set()。拷贝构造函数将其参数对象的堆内存传递给set()。析构函数还回构造函数或赋值运算符为String对象分配的堆内存。

这些成员函数允许在不同的上下文中使用String对象。客户代码可以将String对象定义为一个没有初始化的变量（调用缺省构造函数），或定义为一个用字符数组的值来初始化的对象（调用转换构造函数），也可以用另一个存在的String对象的数据初始化该String对象（调用拷贝构造函数）。

赋值运算符也很聪明。它支持对象的自我赋值（通过检查this指针是否指向实际参数的单元）。它删除已存在的堆内存，并通过调用set()分配和初始化新的堆内存。它允许客户代码使用多次链式赋值的表达式语法（通过返回目标String对象的引用）。注意，即使该赋值运算符函数体返回的是整个String对象（间接引用this指针），它也只是返回对象的一个引用，而没有进行复制。

```
String& String::operator = (const String& s)
{ if (this == &s) return *this;          // no work if self-assignment
  delete [ ] str;                        // return existing memory
  set(s.str);                            // allocate/set new memory
  return *this; }                        // to support chain assignment
```

程序16-3是String类的完整实现（包括内联成员函数getSize()和reset()）。第一个函数返回一个String对象所能容纳的最大符号数，第二个函数返回指向内部字符串的指针，这样，客户代码（函数printString()和modifyString()）可以初始化指向内部字符串的外部指针。

程序16-3 对指向内部数据的指针使用加1运算符的例子

```
#include <iostream>
using namespace std;

class String {
  int size;                                        // string size
  char *str;                                       // start of internal string
  void set(const char* s);                         // private string allocation
public:
  String (const char* s = "")                      // default and conversion
    { set(s); }
  String (const String &s)                         // copy constructor
    { set(s.str); }
  ~String()                                        // destructor
  { delete [ ] str; }
  String& operator = (const String& s);            // assignment
 int getSize() const;                              // current string length
 char* reset() const; } ;                          // reset to start of string

void String::set(const char* s)
 { size = strlen(s);                               // evaluate size
   str = new char[size + 1];                       // request heap memory
   if (str == 0) { cout << "Out of memory\n"; exit(0); }
   strcpy(str,s); }                                // copy client data to heap

String& String::operator = (const String& s)
{ if (this == &s) return *this;                    // no work if self-assignment
  delete [ ] str;                                  // return existing memory
  set(s.str);                                       // allocate/set new memory
  return *this; }                                   // to support chain assignment

int String::getSize() const                         // no change to String object
{ return size; }

char* String::reset() const                         // no change to String object
{ return str; }                                     // return pointer to start

void printString(const String& data)                // no change to string
{ char *p = data.reset();                           // point to first character
  while (*p != 0)                                   // go until end of characters
  { cout << *p;                                     // print the current character
    ++p; }                                          // point to the next character
  cout << endl; }

void modifyString(const String& data, const char text[])   // bad
{ char *p = data.reset();                           // point to first character
  int len = strlen(text) + 1;                       // set the iteration limit
  for (int i=0;  i < len;  i++)                     // go over each character
  { *p = text[i];                                   // copy the current character
    ++p; } }                                        // point to the next character

int main()
{
  String data = "Hello World!";
  printString(data);                                // good output
  modifyString(data,"How is the weather?");
  printString(data);                                // memory is corrupted
```

```
return 0;
}
```

这两个客户函数使用加1运算符来获取并替换其参数String对象中的符号。printString()中的循环一直执行，直到在String参数的内部字符串中找到了终止符0为止（指向内部字符串的指针p必须进行间接引用）。modifyString()中的循环一直执行到参数字符数组text[]的所有字符被拷贝完为止。

main()函数创建并初始化一个String对象，并对其内容进行打印、修改、再打印。由于modifyString()中的循环并不考虑分配给String参数的堆内存的当前大小，这将会导致内存讹用。程序的输出结果如图16-3所示。

```
Hello World!
How is the weather?
```

图16-3　程序16-3的输出结果

这种内存讹用的问题改正起来并不太困难，客户代码modifyString()应对有效空间进行检查，当空间不足时应停止将数据压给对象。

```
void modifyString(const String& data, const char text[])   // ok
{ char *p = data.reset();                  // point to first character
  int len = strlen(text) + 1;              // set one the iteration limit
  int size = data.getSize();               // set another iteration limit
  for (int i=0; i<len && i<size; i++)      // go over each character
  { *p = text[i];                          // copy the current character
    ++p; } }                               // point to the next character
```

该modifyString()函数消除了内存讹用的问题，但仍没有消除设计上的瑕疵：理解服务器类（这里是String）设计细节的责任上托给了客户代码，而没有推向服务器类。这个问题可使用重载运算符函数来解决。

String类的设计也应反映这种态度的改变。为了防止客户滥用其对象，String类应维护对象的状态。在这种情况下，状态应包括指向当前在打印或修改字符的指针。

这是用C++进行设计的一种重要方法。当决定了类所要维护的数据后，应总是包括那些为客户代码反映对象状态的数据成员（依赖客户代码设计者的好心是不够的）。这样使得对象不依赖于客户代码（在程序16-3中我们没有这样做）。下面是String类的更好的修改版本。

```
class String {
  int size;                                // string size
  char *str;                               // start of internal string
  char *ptr;                               // pointer to current symbol
  void set(const char* s);                 // private string allocation
public:
  String (const char* s = "")              // default and conversion
    { set(s); }
  String (const String &s)                 // copy constructor
    { set(s.str); }
  ~String()                                // destructor
  { delete [ ] str; }
 String& operator = (const String& s);     // assignment
 char* operator++();                       // prefix increment operator
```

```
int getSize() const;                    // current string length
char* reset(); } ;                      // no const: object changes
```

这里，ptr指针指向当前符号。这个指针在递增运算符中沿着堆内存移动，该递增运算符返回当前符号的地址供客户访问或修改。加1运算符检查客户的要求是否导致ptr指针超出堆字符数组。如果是，运算符不使指针加1，而是将指针所指向的字符设置为“\0”，以保证字符数组总是正常终止。我们再次使用了ptr-str<size表达式，但这并不是意味着数据成员ptr的地址值比数据成员str的地址值大，只是一种表达指针位移的好方法，不要陷入物理存储管理的细节。

```
char* String::operator ++()             // increment then access
{ if (ptr-str < size)                   // check if room is available
    return ++ptr;                       // pointer to next character
  else
    { *ptr = 0;                         // set the terminating zero
      return ptr; } }                   // do not move it if at end
```

最好在成员函数reset()中初始化该指针。在下一次迭代文本之前，由客户代码调用该函数。这种设计与程序16-3中的设计的重要区别是：在这里，reset()函数不能定义为const——它要修改对象的状态。要经常注意方法行为的模式，不要忽略对成员函数合适地标记（这个问题和非成员函数无关）。

```
char* String::reset()                   // no const: object changes
{ ptr = str;                            // set current pointer to start
  return str; }                         // return pointer to start
```

如果在对象创建时没有将数据成员初始化为特定的值，一些程序员会觉得不舒服。这些程序员也会在每个构造函数的调用中初始化数据成员ptr。在本设计中，每个构造函数调用私有函数set()来初始化数据成员。

```
void String::set(const char* s)
   { size = strlen(s);                  // evaluate size
     str = new char[size + 1];          // request heap memory
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s);                     // copy client data to heap
     ptr = str;    }                    // initialize running pointer
```

这个设计达到什么效果呢？对象一创建就为迭代做好了准备，不再需要显式地发送reset()消息给对象。这是个很好的理念：任务从客户代码（在前一个设计中，客户代码需要调用reset()方法）转移到了对象的构造函数中。

该设计思想适用于对String对象数据的只读访问。遇到终止符0时，printString()函数停止迭代，加1运算符意识到该问题后将指针指向字符串的开始处。

```
char* String::operator ++()             // increment then access
{ if (ptr-str < size)                   // check if room is available
    return ++ptr;                       // pointer to next character
  else
    { *ptr = 0;                         // set the terminating zero
      ptr = str;                        // point to start of data again
      return ptr; } }                   // do not move it if at end
```

这样不再需要reset()成员函数，同时客户代码也不需要调用该函数。每一次扫描

String对象的数据后，指针将复位，并且该对象便可以准备下一次扫描。

```
void printString(String& data)      // no const: string changes
{ char *p = data.reset();           // is this call really needed?
  while (*p != 0)                   // go until end of characters
  { cout << *p;                     // print the current character
    p = ++data; }                   // nice syntax: object changes
  cout << endl; }
```

但在我们的设计中，仍然需要reset()函数，因为客户函数modifyString()可能试图访问超出合法范围的String数据。由于加1运算符不知道客户代码拷贝给字符串的新数据的长度，因而不能将指针移到数据的开始处。

当然，所有这些问题的根源在于modifyString()函数，该函数不管是否存在有效空间都将数据压给其参数String对象。

```
void modifyString(String& data, const char text[])      // no const
{ char *p = data.reset();           // point to first character
  int len = strlen(text) + 1;       // set the iteration limit
  for (int i=0;  i < len;  i++)     // go over each character
  { *p = text[i];                   // copy the current character
    p = ++data; } }                 // point to the next character
```

注意，参数对象data没有const修饰符，但这不是因为modifyString()函数要把新的内容写到它的堆内存中。程序16-3中的modifyString()函数的功能相同，但它的参数标记为const。C++不能记住对象堆内存的改变情况，但对对象数据成员的改变很敏感。加1运算符（与reset()函数）导致数据成员ptr的改变，因此String参数不能使用const修饰符。

程序16-4是为String类实现加1运算符的完整程序。其输出结果如图16-4所示，正如我们看到的，内存讹用问题不复存在。

程序16-4　使用加1运算符作为消息发送给对象的例子

```
#include <iostream>
using namespace std;

class String {
  int size;                                    // string size
  char *str;                                   // start of internal string
  char *ptr;                                   // pointer to current symbol
  void set(const char* s);                     // private string allocation
public:
  String (const char* s = "")                  // default and conversion
    { set(s); }
  String (const String &s)                     // copy constructor
    { set(s.str); }
  ~String()                                    // destructor
  { delete [ ] str; }
  String& operator = (const String& s);        // assignment
 char* operator++();                           // prefix increment operator
 int getSize() const;                          // current string length
 char* reset(); } ;                            // no const: object changes

void String::set(const char* s)
```

```
    { size = strlen(s);                                  // evaluate size
      str = new char[size + 1];                          // request heap memory
      if (str == 0) { cout << "Out of memory\n"; exit(0); }
      strcpy(str,s);                                     // copy client data to heap
      ptr = str;    }                                    // initialize running pointer

String& String::operator = (const String& s)
{ if (this == &s) return *this;                          // no work if self-assignment
  delete [ ] str;                                        // return existing memory
  set(s.str);                                            // allocate/set new memory
  return *this; }                                        // to support chain assignment

int String::getSize() const                              // no change to String object
{ return size; }
char* String::reset()                          // no const: object changes
{ ptr = str;                                   // set current pointer to start
  return str; }                                // return pointer to start

char* String::operator ++()                    // increment then access
{ if (ptr-str < size)                          // check if room is available
    return ++ptr;                              // pointer to next character
  else
     { *ptr = 0;                               // set the terminating zero
       return ptr; } }                         // do not move it if at end

void printString(String& data)                 // no const: string changes
{ char *p = data.reset();                      // point to first character
  while (*p != 0)                              // go until end of characters
  { cout << *p;                                // print the current character
    p = ++data; }                              // point to the next character
  cout << endl; }

void modifyString(String& data, const char text[])
{ char *p = data.reset();                      // point to first character
  int len = strlen(text) + 1;                  // set the iteration limit
  for (int i=0;  i < len;  i++)                // go over each character
  { *p = text[i];                              // copy the current character
    p = ++data; } }                            // point to the next character

int main()
{
  String data = "Hello World!";
  printString(data);                           // good output
  modifyString(data,"How is the weather?");
  printString(data);                           // memory is NOT corrupted
  return 0;
  }
```

```
Hello World!
How is the w
```

图16-4 程序16-4的输出结果

当然，使用加1运算符可以使程序的语法结构变得优美简洁，因此经常使用。这里对这个例子还有两点附带的说明：首先，重载的加1运算符对下标的边界条件进行了检查，这是内部

数据类型中的加1运算符所不能处理的。在运算符函数中这样做的好处是将任务推向服务器类。但是，是这个边界检查使本例优美，而不是加1运算符本身考虑到了printString()和modifyString()中的优美语法。如果函数的名字不使用operator++()而使用movePointer()或next()，也同样要进行边界条件检查。

其次，该加1运算符返回指向堆内存中当前字符的指针，并且让客户代码用该指针做想要做的处理。这样很危险，而且比较容易出错。稍后，我们将介绍另一种比较好的解决方案。

减1运算符和加1运算符很相似，在它的实现过程中没有任何新的原则或概念。

16.2.2 后缀重载运算符

在C++中，内部数据类型的前缀和后缀一元运算符之间的区别是表达式中运算符与操作数的相对位置不同。如果表达式的形式是++data，则该表达式是一个前缀运算；如果表达式的形式是data++，则该表达式是一个后缀运算，区别两者很重要。

对于重载的加1和减1运算符，也同样如此。加1运算符在程序16-4中作为前缀运算符而实现：首先改变目标对象的状态，然后由客户代码返回（当前指针的）新值。

这对于后缀运算符则不恰当。例如对于String类的后缀运算符，它必须首先返回指针的当前值，然后才将该值加1（供以后使用）。因此，该函数不同于程序16-4中的加1运算符，应作为一个单独的函数来实现。

而如何命名这个函数呢？根据C++的规则，函数名应该由关键字operator和运算符的符号组成，在这里即为++或者--。看起来重载的后缀加1运算符函数名应该为operator++()。同样地，重载的后缀减1运算符的名字应该为operator--()。

但它们与重载的前缀运算符函数名字完全相同！在同一作用域中存在两个同名的函数是不合法的，除非这些函数有不同的标识，即不同的参数个数或参数类型。

正如我们在前面提到的，程序员不能随意选取重载运算符的参数。如果将二元运算符实现为成员函数，其第一个操作数作为消息的目标，第二个操作数作为消息的参数。如果将一元运算符实现为成员函数，它惟一的操作数应作为消息的目标，这样的重载运算符不能有参数。

因此，我们要在同一类中实现两个同名（如operator++）和同标识（没有参数）的重载运算符。这当然是自找麻烦。编译程序将会抱怨说它无法区别这样的两个函数。

但是，C++程序员当然要求既能实现加1和减1运算符的前缀形式，也能实现它们的后缀形式。为了解决该问题，C++提供补救措施：使用哑元（dummy）整型参数。

```
char* String::operator ++(int)      // access first then increment
{ if (ptr-str < size)               // check if room is available
    return ptr++;                   // pointer to next character
  else
    { *ptr = 0;                     // set the terminating zero
      return ptr; } }               // do not move it if at end
```

哑元参数的角色很有限：它必须告诉编译程序该函数本质上是另一个函数，而不是对无参数重载的加1（或减1）运算符的重定义。另一方面，该参数在函数体内没有任何作用。因此，我们可以省略参数名，因为编译程序不会指出没有指定参数名。

有了这样的两个函数后，编译程序在客户代码中发现++data时，会解释为data.operator++()，执行前缀运算。当编译程序在客户代码中发现data++时，则解释

为data.operator++(0)，执行后缀运算。operator++()的前缀意义和operator++(int)的后缀意义并未受到语言的加强，而是由类设计者负责它们的内容。

在程序16-4中实现重载后缀运算符后即为程序16-5。该后缀运算符在modifyString()中被调用，返回指向堆内存中当前符号的指针。客户代码必须对函数所返回的值进行间接引用才能改变这个符号。这样语法简洁明了，data变量看起来如同指针一样。

```
*data++ = text[i];                    // copy character
```

该版本的程序输出结果当然与程序16-4中的输出结果相同（见图16-4）。

程序16-5 使用前缀和后缀加1运算符的例子

```
#include <iostream>
using namespace std;

class String {
  int size;                                          // string size
  char *str;                                         // start of internal string
  char *ptr;                                         // pointer to current symbol

  void set(const char* s);                           // private string allocation
public:
  String (const char* s = "")                        // default and conversion
    { set(s); }
  String (const String &s)                           // copy constructor
    { set(s.str); }
  ~String()                                          // destructor
  { delete [ ] str; }
  String& operator = (const String& s);              // assignment
 char* operator++();                                 // prefix increment operator
 char* operator++(int);                              // postfix increment operator
 int getSize() const;                                // current string length
 char* reset(); } ;                                  // no const: object changes

void String::set(const char* s)
   { size = strlen(s);                               // evaluate size
     str = new char[size + 1];                       // request heap memory
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s);                                  // copy client data to heap
     ptr = str;   }                                  // initialize running pointer

String& String::operator = (const String& s)
{ if (this == &s) return *this;                      // no work if self-assignment
  delete [ ] str;                                    // return existing memory
  set(s.str);                                        // allocate/set new memory
  return *this; }                                    // to support chain assignment

int String::getSize() const                          // no change to String object
{ return size; }

char* String::reset()                                // no const: object changes
{ ptr = str;                                         // set current pointer to start
  return str; }                                      // return pointer to start

char* String::operator ++()                          // increment then access
{ if (ptr-str < size)                                // check if room is available
    return ++ptr;                                    // pointer to next character
```

```
  else
        { *ptr = 0;                                    // set the terminating zero
       return ptr; } }                                 // do not move it if at end
char* String::operator ++(int)                         // access then increment
{ if (ptr-str < size)                                  // check if room is available
    return ptr++;                                      // pointer to next character
  else
        { *ptr = 0;                                    // set the terminating zero
       return ptr; } }                                 // do not move it if at end

void printString(String& data)                         // no const: string changes
{ char *p = data.reset();                              // point to first character
  while (*p != 0)                                      // go until end of characters
  { cout << *p;                                        // print the current character
    p = ++data; }                                      // point to the next character
  cout << endl; }

void modifyString(String& data, const char text[])
{ data.reset();                                        // point to first character
  int len = strlen(text) + 1;                          // set the iteration limit
  for (int i=0;  i < len;  i++)                        // go over each character
    *data++ = text[i]; }                               // nice syntax: copy character

int main()
{
  String data = "Hello World!";
  printString(data);                                   // good output
  modifyString(data,"How is the weather?");
  printString(data);                                   // memory is NOT corrupted
  return 0;
  }
```

正如前缀加1和减1运算符一样，重载的后缀运算符可以改善程序的外观。但从软件工程角度而言，它们不是很重要。

16.2.3 转换运算符

要将一个类型的值转换为另一个类型的值，可将目标类型名应用到源类型的值（变量名）上。其语法有两种形式：传统的语法和新的函数式语法。在传统语法中，将目标类型名用括号括起来放在源类型值（或变量）的前面。而在函数式语法中，目标类型名作为有一个参数的函数，源类型的值（或变量）作为该函数的实际参数。

```
int x; double y;
x = int ('A');              // function-like syntax; x contains 65
y = (double)x;              // traditional syntax, y contains 65.0
double *p = &y;             // correct pointer type: this is safe
int *q = (int*)p;           // int q points to double y: trouble
```

这两种转换形式的惟一区别在于：传统的语法可以使用任意合法的类型名，而函数式语法则要求用类型名的标识符。因此在上一个例子中有两个关于数值类型的转换形式，但是只有一个指针类型的转换形式。类型名int*是合法的类型名，但不是合法的标识符。

C++允许任意地在任何数值类型之间进行转换。这些转换可以是显式的（使用转换运算符），也可以是隐式的（没有用转换运算符）。

```
int x; double y;
x = 'A';  y = x;              // implicit casts: no problem in C++
```

C++也允许在任意指针（与引用）之间进行转换，包括程序员定义的任意类型。只是这些转换只能是显式转换，而不允许指针之间进行隐式转换。使用隐式转换几乎总是会有问题，例如下面的代码段中，String指针指向一个整数，该String指针能合法地响应任何String消息。这样，指针将误以为整数所占的内存属于一个String对象。由于String对象的第一个数据成员是一个整型变量size，getSize()消息将获取String对象开始时的值，实际上这是整型变量z的值。如果String类的第一个数据成员不是整数类型，这段代码的显示结果是无意义的。

```
int z = 42; String *ptr = (String*) &z;      // asking for trouble
cout <<"Size: " <<ptr->getSize() <<endl;     // it prints 42
```

在下面这段代码中，整型指针指向String对象，该指针将把String对象的内存解释成一个整数。指针获取的值可用在任意需要整型值的表达式中。在这个例子中，String对象的第一个数据成员为整型变量的size，整型指针将获取到这个值。如果String类的第一个数据成员不是整数类型，代码段将打印出毫无意义的信息。

```
int *r; String s("Hello, World!");
r = (int*) &s;
cout << "String: " << 2 + *r << endl;        // it prints 15
```

在程序16-5的最后加入上面的这两段代码后，其输出结果如图16-5所示。正如我们看到的，这些操作正确地解释了String结构的内存布局，获取了String对象的第一个值，它正好是一个整数类型的数据成员。这在C++中是合法的，但从软件工程角度而言，这对于维护者犹如一场恶梦。一旦改变了类定义中数据成员的次序，而不修改相应的程序代码时，输出结果将会大不一样。

```
Hello World!
How is the w
Size: 42
String: 15
```

图16-5 在程序16-5中增加两段代码后的输出结果

允许指针和引用之间随意转换的规则不能扩展到对象上。不能将一个整数转换为程序员定义类的对象，也不能将程序员定义类的对象转换为一个整数或其他的数值类型或另一个程序员定义类的对象。

```
String s;  Account a;  int x;
x = s;  a = x;  s = a;                       // this is nonsense
```

C++允许有继承关系的类的指针或引用之间进行转换。如果转换目标（指针或引用）是公共基类，转换源（值、指针或引用）是从转换目标公共派生的类，则允许它们之间的隐式转换。当基类指针数组指向不同派生类的对象时，这种隐式转换尤为有用，可以不需用显式转换而将这些对象插入到数组中。

将基类指针（或引用）转换为派生类指针（或引用）时必须用显式转换，这与无关类之间

的转换相似。当基类指针（或引用）指向一个派生类对象，而且必须执行定义在派生类中而不是基类中的操作时，这种显式转换非常有用。显式转换表明所请求的操作属于哪个派生类。

不允许将一个基类对象转换为派生类对象，这与对待无关类型的对象的情况相同。如果必须要实施这样的转换，要在派生类中增加一个转换构造函数。该构造函数应该有一个基类类型的参数。我们在第15章中有这样的转换和构造函数的例子。

对于通过继承相关的类，C++也允许违反强类型规则的另一种情况。虽然不允许将基类对象隐式地转换为派生类对象，但允许将派生类对象隐式地转换为基类对象。如果不显式转换，派生类对象中多余的数据成员和操作将会丢掉。

总之，C++只是对程序员派生的类的对象保持了强类型规则，对于其他类型则允许转换。有些转换只能使用转换运算符显式进行（对于任意类型的指针和引用）；其他转换则可以隐式地进行，不需要显式转换（数值类型之间或将派生类的对象、指针和引用转换为基类的对象、指针和引用）。这些转换为程序员实现不同类型的值的处理算法提供了很大的灵活性。由于这些转换破坏了强类型规则，所以去掉了语法检查给我们提供的保护。

不仅如此，C++还允许程序员对定义类的对象实现其他的转换，而这正是强类型规则保持的惟一种类。注意语法检查提供的保护并没有轻易地消除，只是对于指定的类消除而已。程序员通过使用转换构造函数和转换运算符指定这些被消除保护的类。

我们在第9章中描述了转换构造函数。该转换构造函数有一个类型参数，要将该类型的参数转换为某给定的类。例如，String类有一转换构造函数将字符数组的值转换为String类型的值。

```
String (const char* s)              // conversion constructor
  { set(s); }
```

有了这个构造函数后，可以在需要String对象的地方使用字符数组，而不会产生编译时语法错误。由于String是程序员定义的类，使用String对象的地方不会很多，主要是在对象定义、按值（而不是按指针或引用）传递参数、赋值以及将消息发送给对象的时候。

```
printString("Hi there");                  // error: pass by reference
printString(String("Hi there"));          // OK: object is created
int sz = String("Hi there").getSize();    // object is created
```

如果编译程序能确认所需类型的标识，则可进行隐式转换（即不使用转换运算符）。如下面的赋值语句：

```
String s;
s = "Hi there";           // same as s = String("Hi there");
```

在所有这些情况下，在栈中创建了一个未命名的String对象，并调用转换构造函数。然后（调用析构函数）删除该对象。C++并没有指定对象销毁的确切时间。编译程序的编写者必须保证对象在使用后瞬间内仍然存在，并在当前作用域结束前消失。

通常假设，转换构造函数的参数应该与该类的某个数据成员的类型相同。例如，String类的转换构造函数有一个字符数组（字符指针）参数，而String有一个字符指针数据成员。这通常是正确的，但是并不是必需的。

类设计者可以用他认为合适的任意类型作为转换构造函数的参数。例如，程序16-1中的Account类有一个转换构造函数，其参数为String类型，但Account类中没有String类

的数据成员。下面是该构造函数：

```
Account(String& s)                          // conversion (String changes)
{ char* p = s.reset();                      // get pointer to the array
  owner = new char[strlen(p)+1];            // allocate heap memory
  if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
  strcpy(owner, p);                         // initialize data fields
  balance = 0; }                            // default for new account
```

这样，在需要使用Account对象的任何地方都可以用String对象。一个收支情况为0的新的Account对象没什么作用，除非它是一些消息的目标，因此，当用String对象而不是字符数组表示拥有者数据时，最好使用该构造函数来创建Account对象。

```
String owner("Smith");
Account a(owner);                       // create and initialize
a += 500;                               // use the Account object
```

可以看到，转换构造函数允许类设计者在需要某给定类的数据时，显式指定所能使用的类型。注意这些构造函数实现了对象之间的转换，而不是C++中一直允许的指针或引用之间的转换。转换构造函数所实现的转换可以是显式的（如果编译程序不能从类执行转换的上下文环境中进行定义），甚至也可以是隐式的（如果转换的目标在上下文环境中很明显）。

转换构造函数削弱了强类型规则，当转换被无意执行时，失去了编译程序的保护。然而，在C++中转换构造函数很流行，因为它们为编写C++代码提供了更大的灵活性。

指定对象之间可以进行哪些转换的第二种机制是使用转换运算符。转换运算符是一个重载运算符，其名字为目标类型的名字。作为一个重载运算符，转换运算符应遵循一般的重载运算符规则。但其语法也有独特之处。

与构造函数和析构函数相似，它不应该有返回类型。与析构函数相似，它不应该有参数。与构造函数和析构函数不同的是，它必须返回一个值，该返回值应属于转换后的类型（即运算符名字所表示的类型）。下面是String类的一个整型转换运算符。

```
String::operator int() const            // no change to String object
{ return size; }                        // no return type, just value
```

通常，返回值是转换运算符所属类的某个合适类型的数据成员的值。如果该类有多个该类型的数据成员，则出类的设计者决定在转换中使用哪个数据成员的值更为合适。如果无法决定选择哪个域，也不要因此而感到痛苦。我们要记住使用运算符是为客户代码提供方便和简洁的语法，而不是处理一般成员函数所不能处理的事。

根据情况的不同，一个类可以有多个转换运算符。下面是String类的字符指针转换运算符，它可以代替reset()方法。

```
String::operator char* () const         // object does not change
{ return str; }                         // return pointer to start
```

现在可以用这两个String转换运算符精简客户代码（Account转换构造函数）。

```
Account(const String& s)
{ int len = (int)s;                     // get the size of string
  owner = new char[len+1];              // allocate heap memory
  if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
  strcpy(owner, (char*)s);              // initialize data fields
  balance = 0; }
```

在这里，显式转换可以帮助维护者理解函数中的值流，但它们并不是必需的。如果编译程序不难理解所需的是什么类型，则可以省略显式转换，用隐式转换就可以了。

```
Account(const String& s)
{ int len = s;                          // implicit cast to integer
  owner = new char[len+1];              // allocate heap memory
  if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
  strcpy(owner, s);                     // implicit cast to char array
  balance = 0; }
```

正如我们看到的，在任何需要整型或字符数组值的地方都可以使用String对象。要记住：在C++程序中，不论显式转换还是隐式转换都是消息，即调用重载转换运算符的函数。编译程序将把上面两个版本的构造函数解释为如下形式：

```
Account(const String& s)
{ int len = s.operator int();               // call to an operator
  owner = new char[len+1];                  // allocate heap memory
  if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
  strcpy(owner, s.operator char*());        // call to an operator
  balance = 0; }
```

在大多数情况下，转换运算符用于从对象中抽取某个域的值。但这并不是其惟一用法。与转换构造函数相似，类的设计者还可以用转换运算符来指明类对象的转换类型，无论设计者指定什么作为合法的转换，都是可行的。例如，Account类可能支持两个转换运算符：转换为double和String，即使Account类没有String类型的数据成员。

```
Account::operator double () const           // object does not change
{ return balance; }                         // return double value

Account::operator String () const           // create a String object
{ return owner; }                           // implicit conversion
```

我们要注意在第二个转换运算符中Account类对象隐式地转换为String类。创建了String对象，并将它返回给客户代码使用，且在客户作用域中自动撤销。注意我们不是说“在客户代码结束时”，因为撤销的时间并没有确切地定义。还要注意在运算符名字中使用引用会出现语法错误，因为在C++中所有的引用都必须是常量。

```
Account::operator String& () const          // syntax error
{ return owner; }                           // implicit conversion
```

为了解决这个问题，可以定义运算符名字为常量String引用。这样就可以通过编译了。

```
Account::operator const String& () const    // no syntax error
{ return owner; }                           // not a good idea
```

这样是不错，但所返回的引用是指向未命名的临时对象，在编译程序希望的任何时候都可以撤销该对象。因此，客户代码可能接收到一个无效的引用。这不是一个好的程序设计实践。

有了这些转换，客户代码可将String对象转换为一个整数值或一个字符指针（通过调用转换运算符），还可转换为一个Account对象（通过调用Account转换构造函数）。还可将一个Account对象转换为一个double值或一个String值（通过调用转换运算符）。而且，一个字符数组也可被转换为一个String对象（通过调用String转换构造函数）。

程序16-6演示了这些转换。客户代码处理String对象时就像在处理字符数组一样，处

理Account对象也如同是处理double数据值或String数据值。程序的执行结果如图16-6所示。

程序16-6 使用转换构造函数和转换运算符的例子

```
#include <iostream>
using namespace std;

class String {
  int size;                                          // string size
  char *str;                                         // start of internal string
  void set(const char* s);                           // private string allocation
public:
  String (const char* s = "")                        // default and conversion
    { set(s); }
  String (const String &s)                           // copy constructor
    { set(s.str); }
  ~String()                                          // destructor
  { delete [ ] str; }
  String& operator = (const String& s);              // assignment
  operator int() const;                              // current string length
  operator char* () const;                           // return pointer to start
} ;
void String::set(const char* s)
   { size = strlen(s);                               // evaluate size
     str = new char[size + 1];                       // request heap memory
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s); }                                // copy client data to heap

String& String::operator = (const String& s)
{ if (this == &s) return *this;                      // no work if self-assignment
  delete [ ] str;                                    // return existing memory
  set(s.str);                                        // allocate/set new memory
  return *this; }                                    // to support chain assignment

String::operator int() const                         // no change to String object
{ return size; }

String::operator char* () const                      // object does not change
{ return str; }                                      // return pointer to start

class Account {                                      // base class of hierarchy
protected:
  double balance;                                    // protected data
  char *owner;
public:

  Account(const char* name, double initBalance)      // general
  { owner = new char[strlen(name)+1];                // allocate heap space
    if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
       strcpy(owner, name);                          // initialize data fields
       balance = initBalance; }

  Account(const String& s)
  { int len = s;                                     // get the size of string
    owner = new char[len+1];                         // allocate heap memory
    if (owner == 0) { cout << "\nOut of memory\n";  exit(0); }
    strcpy(owner, s);                                // initialize data fields
```

```
    balance = 0; }

  operator double () const                          // object does not change
  { return balance; }

  operator String () const                          // create a String object
  { return owner; }                                 // implicit conversion

  void operator -= (double amount)
  { balance -= amount; }                            // pop responsibility up

  void operator += (dcuble amount)
  { balance += amount; }                            // increment unconditionally
} ;
int main()
{
  String owner("Smith");                           // conversion constructor
  Account a(owner);                                // conversion constructor
  a += 500;  a -=200;  a += 400;                   // overloaded operators
  String s = a;                                    // handle as a String value
  double limit = 2 * a;                            // handle as a double value
  cout << "Name: " << (char *)s << endl;           // explicit conversion
  cout << "Balance: " <<(double)a <<endl;          // explicit conversion
  cout << "Credit limit: " << limit << endl;
  return 0;
  }
```

```
Name: Smith
Balance: 700
Credit limit: 1400
```

图16-6　程序16-6的输出结果

如果在一个给定的上下文中可以使用几种不同类型，需要提示编译程序使用哪种类型。可以以转换的形式给出这个提示。在输出语句中，任意类型的值都是合法的输出值。如果有多种转换可能，则必须进行显式转换。例如，上面的String数据值可以转换为一个整数和一个字符数组。编译程序（和维护人员）就需要获知程序员希望进行哪种转换。

如果没有找到指定类型的转换，编译程序将查找内部数据类型的（数值类型之间的）转换，以使这种调用成为可能。例如以下语句：

```
cout << "Balance: " <<(float)a <<endl;      // explicit conversion
```

Account类并未提供float转换运算符，但并不意味着这个语句是错误的。由于在内部数据类型中可以将double转换为float，编译程序将Account数据值转换为double，再将double转换为float。编译程序不能连续进行多个程序员定义的转换，也不能将多个内部数据类型的转换和程序员定义的转换连接在一起。但是一个程序员定义的转换和一个内部数据类型的转换的连接是可行的。

16.3　下标和函数调用运算符

这两个运算符是二元运算符，但它们与其他的C++二元重载运算符有明显不同，其不同在

于客户对运算符的调用转换成二元表达式的方式。对于其他的重载运算符，消息的目标作为表达式的左操作数，函数调用的参数作为表达式的右操作数，将运算符（从方法名得到）放在右操作数和左操作数之间。而对于下标和函数调用运算符则不是这样（很快将在下面的例子中看到）。

这两个重载运算符与其他的重载运算符的另一个区别是：这两个重载运算符不能实现为全局非成员函数。其目的是让C++编译程序能更加简单地分析上下文。

16.3.1 下标运算符

理想情况下，重载的下标运算符的表达式形式应与内部数据类型的下标运算符语法一样：即变量名后加上左、右括号括起来的下标。例如，s[i]被解释为下标i应用到对象（变量）s上。

当然，操作所代表的意义可以随意指定。大多数C++程序员（及所有的C++库）将该表达式解释为：获取对象s的第i个元素的值。另一个流行的解释是赋值给对象s的第i个元素。

这两种解释都认为对象s是一个容器，容纳一个数组、一个链表或其他合适的成员的集合，而s[i]表达式指的是容器中第i个元素的值。因此，重载下标运算符是一个返回容器中某个元素值的函数。

作为容器类的简单例子，我们考虑一个简化了的Array类。这是一个与程序16-6中的String类相似的容器类。容器的元素是int类型而不是char类型。

该Array类完善了C++数组类型中的两个不足：数组溢出和无效的下标值。第一个问题可通过在堆中分配数组元素来解决，但这个办法与许多方法一样将带来其他的问题：如程序的完整性问题。为了保护程序的完整性，Array类应提供一个拷贝构造函数、一个析构函数和一个重载的赋值运算符。

第二个无效下标的问题可提供成员函数getInt()和setInt()来解决，这两个成员函数代表客户代码访问内部的Array内存。

```
class Array {
public:
  int size;                                  // number of valid components
  int *ptr;                                  // pointer to array of components
  void set(const int* a, int n);             // allocate/init heap memory
public:
  Array (const int* a,int n);                // general constructor
  Array (const Array &s);                    // copy constructor
 ~Array();                                   // return heap memory
  Array& operator = (const Array& a);        // copy array to another
  int getSize() const;
  int getInt(int i) const;                   // return the i-th component
  void setInt(int i, int x);                 // set int x at position i
 } ;
```

也许没有必要解释，但仍然值得一提的是第i个位置实际上是第（i+1）个位置。也就是说，第1个元素的下标是0，第2个元素的下标是1，依次类推。

这样，成员函数getInt()返回的是内部堆数组中的第i个下标所对应的值。由于getInt()称为函数，我们除了从数组中获取值之外，还可以做其他有用的事情，例如检查下标对于字符串边界的有效性。

```
int Array::getInt (int i) const     // object does not change
{ if (i < 0 || i >= size)           // index is out of bounds
    return ptr[size-1];             // return the last component
  return ptr[i]; }                  // legal index: return value
```

使用这个函数，客户代码可以实现与C++基本的数组类似的迭代算法，但是因为该函数不能访问数组外的内存区域，从而更加安全一些。

```
void printArray(const Array& a)
{ int size = a.getSize();                  // get array size
  for (int i=0;  i < size;  i++)           // go over each component
  { cout << " " << a.getInt(i); }          // print next component
  cout << endl << endl; }
```

在getInt()函数中增加这个功能后带来的一个问题是在一定程度上减慢了程序的执行速度。C和C++没有引入下标检查的原因就是想避免这种速度的减慢。但是大多数现代应用程序都可以接受这个速度变慢的问题。如果这个问题变得很重要，我们也可以总是使用一个执行速度更快的函数，不会在检查下标上消耗时间，而且是内联地实现。

该设计的另一个问题是强迫客户代码检查返回值（不管客户代码是否需要这样做）。毕竟，如果客户代码觉得需要进行检查，可使用getSize()成员函数获取Array对象数据的大小（就像上面的printArray()函数一样，或者像程序16-6中相似的容器String类做的那样）。这就允许客户代码显式地进行检查，这个目的是合法的。然而每一个设计的决策都是权衡后的结果。一般来说，将任务（包括完整性检查）推到服务器代码，简化客户代码以让它的算法不受完整性检查等细节的困扰，这些都被认为是合理的软件工程实践。

当下标无效时返回容器中的最后一个值，这似乎是一个好的决策。另一个方法是返回特定的岗哨值，例如0。这可以让客户代码构造对Array对象的迭代，并在返回0代码时终止迭代。但是只有当从应用程序的角度来看组件的0值是非法时，这个方法才可行，但是0值通常是合法的。

下面，我们讨论setInt()方法。在执行此方法时进行边界检查也是一个好主意。

```
void Array::setInt(int i,int x)     // modify Array object
{ if (i < 0 || i >= size)           // check if index is legal
    return;                         // no op if it is out of bounds
  ptr[i] = x; }                     // legal index: set the value
```

有人认为边界检查对于该函数比在getInt()函数中更为重要。在getInt()中，可以冒险将不正确的数据传送给客户代码，在调试和测试时会发现。而在setInt()中，如果不进行边界检查将有可能导致内存滥用，而且在早期检查中不会发现。

对那些仍然饱受严格的C++下标之苦的人来说，下面版本中的两个函数允许客户代码将下标的改变范围从1到数组的大小值。

```
int Array::getInt (int i) const     // object does not change
{ if (i < 1 || i > size)            // index is out of bounds
    return ptr[size];               // return the last component
  return ptr[i-1]; }                // legal index: return value

void Array::setInt(int i,int x)     // modify Array object
{ if (i < 1 || i > size)            // check if index is legal
    return;                         // no op if it is out of bounds
```

```
  ptr[i-1] = x; }                    // legal index: set the value
```

下面的printArray()充分利用了上面的设计。希望到此为止，大家都会熟悉通用的规范而不必写出这样的代码。

```
void printArray(const Array& a)
{ int size = a.getSize();                  // get array size
  for (int i=1;  i <= size;  i++)          // go from 1 to size
  { cout << " " << a.getInt(i); }          // print next component
  cout << endl << endl; }
```

程序16-7是Array类及其客户代码的实现，程序的输出结果如图16-7所示。

程序16-7 使用Array类作为整型成员的容器

```
#include <iostream>
using namespace std;

class Array {
public:
  int size;                                // number of valid components
  int *ptr;                                // pointer to array of components
  void set(const int* a, int n);           // allocate/init heap memory
public:
  Array (const int* a,int n);              // general constructor
  Array (const Array &s);                  // copy constructor
 ~Array();                                 // return heap memory
  Array& operator = (const Array& a);      // copy array to another
  int getSize() const;
  int getInt(int i) const;                 // return the i-th component
  void setInt(int i, int x);               // set int x at position i
 } ;

 void Array::set(const int* a, int n)
 { size = n;                               // evaluate array size
   ptr = new int[size];                    // request heap memory
   if (ptr == 0) { cout << "Out of memory\n"; exit(0); }
   for (int i=0;  i < size;  i++)
       ptr[i] = a[i]; }                    // copy client data to heap

Array::Array (const int* a, int n)         // general
 { set(a,n); }

Array::Array (const Array &a)              // copy constructor
 { set(a.ptr,a.size); }

Array::~Array()                            // destructor
 { delete [ ] ptr; }

Array& Array::operator = (const Array& a)
{ if (this == &a) return *this;            // no work if self-assignment
  delete [ ] ptr;                          // return existing memory
  set(a.ptr,a.size);                       // allocate/set new memory
  return *this; }                          // to support chain assignment

int Array::getSize() const                 // get array size
{ return size; }
```

```
int Array::getInt (int i) const                    // object does not change
{ if (i < 0 || i >= size)                          // index is out of bounds
    return ptr[size-1];                            // return the last component
  return ptr[i]; }                                 // legal index: return value
void Array::setInt(int i,int x)                    // modify Array object
{ if (i < 0 || i >= size)                          // check if index is legal
    return;                                        // no op if it is out of bounds
  ptr[i] = x; }                                    // legal index: set the value

int main()
{
  int arr[] = { 1,3,5,7,11,13,17,19 } ;            // data to process
  Array a(arr, 8);                                 // create the object
  int size = a.getSize();                          // get array size
  for (int i=0;  i < size;  i++)                   // go over each component
  { cout << " " << a.getInt(i); }                  // print next component
  cout << endl << endl;
  for (int j=0;  j < size;  j++)                   // go over the array again
   { int x = a.getInt(j);                          // get next component
     a.setInt(j, 2*x); }                           // update the value
  for (int k = 0;  k < size;  k++)
  { cout << " " << a.getInt(k); }                  // print updated array
  cout << endl;
  return 0;
  }
```

```
1 3 5 7 11 13 17 19

2 6 10 14 22 26 34 38
```

图16-7　程序16-7的输出结果

在这个例子中，函数set()、构造函数、析构函数及赋值运算符与程序16-6中String类的成员函数相似。其主要区别是String函数在其循环中使用了终止符0，而Array函数使用了容器中的元素个数。

一个完善的Array类应能将新的成员增加到数组末尾或中间，能删除成员，能进行成员比较，测试有效数据是否存在等。为了简单起见，我们省略了这些功能。

我们曾提到过，使用getInt()方法的语法很好，也与C++中基本的数组类型非常相似。而使用setInt()方法的语法则有些糟糕。

```
for (int j=0;  j < size;  j++)      // go over the array again
 { int x = a.getInt(j);             // get next component
   a.setInt(j, 2*x); }              // update the value
```

这里，我们遍历数组中的每个成员，并将每个成员的值翻一倍。修改成员的语法不同于访问成员的语法。另一方面，基本C++数组访问数组元素的语法（如x=a[j]）与为数组元素赋值的语法（如a[j]=2*x）相同。

让客户代码更新容器中值的方式与访问值的方式一样，这样将更好。

```
  for (int j=0;  j < size;  j++)          // go over the array again
   { int x = a.getInt(j);                 // get next component
//     a.setInt(j, 2*x); }                // update the value
```

```
    a.setInt(j) = 2 * x; }                    // update the value
```

在传统的程序设计中，这是不可能的——函数的返回值不能用在赋值运算的左边。但在C++中，如果函数返回的是值的引用而不是值本身，则是允许的。当然，返回的引用必须是有效的引用，不能在函数终止时消失，但这是另外一个问题。

在第7章中，我们已介绍了从函数中返回引用以编写简洁和富于表现力的客户代码。在这里，我们再次应用它。下面我们删除setInt()界面中的值参，将setInt()的返回类型由整型值改为整型引用。

```
int& Array::setInt(int i)              // modify Array object
{ if (i < 0 || i >= size)              // check if index is legal
    return ptr[size-1];                // return the last component
  return ptr[i]; }                     // legal index: return reference
```

该函数支持上面的客户循环：它返回一个整数的引用，循环向引用正指向的地址赋值。该方案的关键在于引用指的不是一个在setInt()函数终止时会消失的局部值，而是一个数组元素，它在调用setInt()之前已存在，而且在setInt()终止后仍将继续存在。

下面我们将getInt()与新版本的setInt()做一下比较，可以看出它们的实现相同。客户代码需要两个函数吗？两个函数之间在函数接口上有两个不同之处。getInt()的返回值是一个值，而不是引用。这个问题并不严重，让我们将getInt()的返回值改为指向整数的引用。

```
int& Array::getInt(int i) const        // object does not change
{ if (i < 0 || i >= size)              // index is out of bounds
    return ptr[size-1];                // return the last component
  return ptr[i]; }                     // legal index: return reference
```

有了这个函数后，程序16-7中客户代码仍可像以前一样工作。

```
for (int i=0;  i < size;  i++)         // go over each component
{ cout << " " << a.getInt(i); }        // OK if reference is returned
cout << endl << endl;
for (int j=0;  j < size;  j++)         // go over the array again
 { int x = a.getInt(j);                // OK if reference is returned
   a.setInt(j) = 2 * x; }              // OK if reference is returned
```

第二个区别是getInt()并不能改变它操作的对象的状态，该对象标注为const。另一方面，setInt()修改了它作为消息传递的目标对象的状态，因此没有标注为const。

在处理const修饰符时许多C++程序员都会犯这个典型的错误。的确，setInt()函数修改了属于目标对象的堆内存的状态，但该内存不是对象的一部分，它只是属于对象而已。数据成员才是对象的一部分，而不是堆内存。函数setInt()并未修改目标对象的数据成员，这是最关键的。这是C++程序员必须时时记住的概念之一。

我们错误地设计了setInt()函数，它应该标注为const，因为它并未修改目标对象的状态。

```
int& Array::setInt(int i) const        // Array object is not modified
{ if (i < 0 || i >= size)              // check if index is legal
    return ptr[size-1];                // return the last component
  return ptr[i]; }                     // legal index: return reference
```

从以上版本的函数来看就很清楚了。但是即使从程序16-7中的函数版本来看也应该是清楚的。虽然注释说对象改变了，但是程序16-7中的setInt()并没有修改它的目标对象。一定要考虑const修饰符的作用。

现在，getInt()和setInt()函数看起来完全一样了，我们可以去掉其中一个。对程序16-7进行修改，只使用这两个函数中的一个，即getInt()，修改后的程序为程序16-8。该程序的输出结果与程序16-7的输出结果相同。

程序16-8　使用同一成员函数获取并设置Array数据

```
#include <iostream>
using namespace std;

class Array {
public:
  int size;                                    // number of valid components
  int *ptr;                                    // pointer to array of integers
  void set(const int* a, int n);               // allocate/init heap memory
public:
  Array (const int* a, int n);                 // general constructor
  Array (const Array &s);                      // copy constructor
 ~Array();                                     // return heap memory
  Array& operator = (const Array& a);          // copy array
  int getSize() const;
  int& getInt(int i) const;                    // get/set value at position i
} ;

 void Array::set(const int* a, int n)
 { size = n;                                   // evaluate array size
   ptr = new int[size];                        // request heap memory
   if (ptr == 0) { cout << "Out of memory\n"; exit(0); }
   for (int i=0;  i < size;  i++)
        ptr[i] = a[i]; }                       // copy client data to heap

Array::Array(const int* a, int n)              // general constructor
 { set(a,n); }

Array::Array (const Array &a)                  // copy constructor
 { set(a.ptr,a.size); }

Array::~Array()                                // destructor
 { delete [ ] ptr; }

Array& Array::operator = (const Array& a)
{ if (this == &a) return *this;                // no work if self-assignment
  delete [ ] ptr;                              // return existing memory
  set(a.ptr,a.size);                           // allocate/set new memory
  return *this; }                              // to support chain assignment

int Array::getSize() const                     // get array size
{ return size; }

int& Array::getInt(int i) const                // Array object is not modified
{ if (i < 0 || i >= size)                      // check if index is legal
    return ptr[size-1];                        // no op if it is out of bounds
  return ptr[i]; }                             // legal index: set the reference
```

```
int main()
{
  int arr[] = { 1,3,5,7,11,13,17,19 } ;     // data to process
  Array a(arr, 8);                          // create an object
  int size = a.getSize();                   // get array size
  for (int i=0;  i < size;  i++)            // go over each component
  { cout << " " << a.getInt(i); }           // print next component
  cout << endl << endl;
  for (int j=0;  j < size;  j++)            // go over the array again
  { int x = a.getInt(j);                    // get next component
    a.getInt(j) = 2*x; }                    // update the value
  for (int k = 0;  k < size;  k++)
  { cout << " " << a.getInt(k); }           // print updated array
  cout << endl;
  return 0;
  }
```

下一步是用重载的下标运算符替换成员函数getInt()。函数本身的修改非常简单，只需要去掉函数名getInt，使用operator关键字，再加上运算符号（这里是[]）。

```
//int& Array::getInt(int i) const      // Array object is not modified
int& Array::operator [](int i) const  // operator header
{ if (i < 0 || i >= size)              // check if index is legal
    return ptr[size-1];                // no op if it is out of bounds
  return ptr[i]; }                     // legal index: set the reference
```

在客户代码中也要做类似的修改，现在成员函数的名字是operator[]，而不是getInt。

```
int size = a.getSize();                     // get array size
for (int i=0;  i < size;  i++)              // go over each component
{ cout << " " << a.operator[](i); }         // print next component
cout << endl << endl;
for (int j=0;  j < size;  j++)              // go over the array again
{ int x = a.operator[](j);                  // get next component
  a.operator[](j) = 2*x; }                  // update the value
for (int k = 0;  k < size;  k++)
{ cout << " " << a.operator[](k); }         // print updated array
cout << endl;
```

当然，我们对程序16-7中第一个实现的修改不会到此为止。还需用表达式语法替换函数调用语法。但是像任何其他运算符那样对待operator[]会产生很糟糕的代码。例如我们是如何对待operator+的呢？我们将消息目标作为第一个操作数，然后是从运算符得到的符号，例如+，最后是第二个操作数。

```
a.operator+(b);                             // same as a + b;
```

如果我们以同样的方式处理下标运算符，就会得到可读性差的代码：

```
cout << " " << a.operator[](i);             // same as a[]i !
```

为了使重载的下标运算符与内部数据类型的下标运算符的使用一致，C++免去了一些特殊的处理。编译程序被告知要容忍脱离通用规则的情况。程序16-9是使用重载下标运算符的例子。

程序16-9　使用重载下标运算符获取并设置Array数据

```
#include <iostream>
using namespace std;

class Array {
public:
  int size;                                    // number of valid components
  int *ptr;                                    // pointer to array of integers
  void set(const int* a, int n);               // allocate/init heap memory
public:
  Array (const int* a, int n);                 // general constructor
  Array (const Array &s);                      // copy constructor
 ~Array();                                     // return heap memory
  Array& operator = (const Array& a);          // copy array
  int getSize() const;
  int& operator [ ] (int i);                   // get/set value at position i
 } ;

 void Array::set(const int* a, int n)
 { size = n;                                   // evaluate array size
   ptr = new int[size];                        // request heap memory
   if (ptr == 0) { cout << "Out of memory\n"; exit(0); }
   for (int i=0;  i < size;  i++)
       ptr[i] = a[i]; }                        // copy client data to heap

Array::Array(const int* a, int n)              // general constructor
 { set(a,n); }

Array::Array (const Array &a)                  // copy constructor
 { set(a.ptr,a.size); }

Array::~Array()                                // destructor
 { delete [ ] ptr; }
Array& Array::operator = (const Array& a)
{ if (this == &a) return *this;                // no work if self-assignment
  delete [ ] ptr;                              // return existing memory
  set(a.ptr,a.size);                           // allocate/set new memory
  return *this; }                              // to support chain assignment

int Array::getSize() const                     // get array size
{ return size; }

int& Array::operator [](int i)                 // Array object is not modified
{ if (i < 0 || i >= size)                      // check if index is legal
    return ptr[size-1];                        // no op if it is out of bounds
  return ptr[i]; }                             // legal index: set the value

int main()
{
  int arr[] = { 1,3,5,7,11,13,17,19 } ;        // data to process
  Array a(arr, 8);                             // create an object
  int size = a.getSize();                      // get array size
  for (int i=0;  i < size;  i++)               // go over each component
//{ cout <<" "<<a.operator[](i); }             // alternative syntax
  { cout << " " << a[i]; }                     // print next component
  cout << endl << endl;
  for (int j=0;  j < size;  j++)               // go over the array again
```

```
    { int x = a[j];                                    // special deal
//    { int x = a.operator[](j);                       // alternative syntax
      a[j] = 2*x; }                                    // special deal
  for (int k = 0;  k < size;  k++)
  { cout << " " << a[k]; }                             // print updated array
  cout << endl;
  return 0;
  }
```

很难说这个版本的程序比程序16-7的最初版本有多少提高。但是运算符的语法是好的。而且无疑有助于复习从函数返回引用而不是值的问题，也可以复习const修饰符的使用。

16.3.2 函数调用运算符

函数调用运算符（在C++中将()也认为是运算符）也可用来访问或设置容器类对象的成员值。当容器在堆内存中的结构是二维数组而不是一维数组（如前一个例子）时常使用该运算符。

使用函数调用运算符而不使用下标运算符的原因是：对于二维数组，C++要连续使用两个下标运算符，例如，m[i][j]。如果按照传统的程序设计语法使用一个下标运算符，如m[i,j]，就会使得下标运算符成为一个三元运算符。（本例中，其操作数为数组m、下标i和j）。对于更多维的数组而言，下标的个数将可能不止两个。

C和C++的设计者认为允许加运算符改变它的元是可行的，一元加和二元加都可以改变。但对于下标运算符来说类似的修改是不允许的。下标运算符是下一个二元运算符，其操作数不能多于两个。

因此，我们使用函数调用运算符来代替下标运算符，其优点主要在于函数调用运算符可以有任意多个参数。

例如，我们考虑一个Matrix类，它实现了一个方阵。客户代码通过定义两个下标来操纵矩阵成员：一个下标表示矩阵的行，另一个表示矩阵的列。可创建Matrix对象，将它作为函数参数传递，也可以相互赋值。这个实现的基础是动态分配的线性数组，数组的大小依赖于方阵的大小。

Matrix类使用了私有函数make()，该函数与前一个例子中的set()函数类似，但它没有初始化堆内存。make()函数被转换构造函数、拷贝构造函数及重载赋值运算符所调用。

```
class Matrix {
  int *cells;                                  // heap array to house the matrix
  int size; // number of rows and of columns
  int* make(int size)                          // private allocator function
  { int* p = new int [size * size];            // total number of elements
    if (p == NULL) { cout << "Matrix too big\n";  exit(0); }
    return p; }                                // return pointer to heap storage
public:
  Matrix (int sz) : size(sz)                   // conversion constructor
  { cells = make(size); }                      // heap memory is not initialized
  Matrix (const Matrix& m) : size(m.size)
  { cells = make(size); }                      // copy constructor: for safety
  Matrix& operator = (const Matrix& m);        // assignment operator
  int getSize() const                          // size of the side
  { return size; }
  int& get (int r, int c) const;               // access or modify a component
```

```
  ~Matrix() { delete [] cells; }              // destructor
  } ;
```

赋值运算符释放已存在的堆内存，在堆中分配新的内存，将参数Matrix对象的数据拷贝给赋值的目标对象。

```
Matrix& Matrix::operator = (const Matrix& m)     // assignment
{ if (this == &m) return *this;                  // no work if self-assignment
  delete [ ] cells;                              // return existing memory
  cells = make(m.size);                          // allocate/set new memory
  size = m.size;                                 // set the matrix size
  for (int i=0;  i<size*size;  i++)              // copy data
        cells[i] = m.cells[i];
  return *this; }                                // to support chain assignment
```

get()函数结合了前面例子中getInt()函数和setInt()函数的功能。它使用调用者传来的行、列坐标（当然是从0开始计数的）计算矩阵单元在线性数组中的位置。如果坐标是非法的，则无警告地返回数组中最后一个元素；如果坐标是合法的，则返回存储在指定坐标处的数据。

```
int& Matrix::get (int r, int c) const
 {  if (r<0 || c<0 || r>=size || c>=size)        // check validity
      return cells[size*size-1];                 // return last matrix cell
    return cells[r*size + c]; }                  // return requested cell
```

如果行或列坐标值超出矩阵范围，返回最后一个矩阵单元并不是最好的办法。另一个可行的办法是终止执行操作，或者返回异常，但是我们不愿意终止操作，而且也没有讨论过异常。还有一个解决方法是返回一个应用程序没有使用的值，例如，最大整数值MAX_INT。但常量不能被按引用返回（以避免我们可能决定修改它）。

```
int& Matrix::get (int r, int c) const            // not a good version
 {  if (r<0 || c<0 || r>=size || c>=size)        // check validity
     return MAX_INT;                             // illegal to return by reference
   return cells[r*size + c]; }                   // return requested cell
```

程序16-10实现了Matrix类及刚才所描述的get()函数。客户函数printMatrix()遍历矩阵的行和列，并依次打印每一行元素。注意使用了setw()操纵符（manipulator），不幸的是<iostream>包含的文件并不支持用户使用操纵符，我们必须包含<iomanip>头文件。

客户函数main()创建了方阵对象，并用行数和列数的乘积初始化了每个单元（按科学计数而不是按C++计数，即从1开始）。在这个循环中，main()函数将get()函数的返回值作为一个左值，然后main()调用了printMatrix()函数，该函数将get()函数的返回值作为其右值。接着，main()将矩阵对角线上的元素置为0（使用get()作为左值），并再次打印矩阵。最后，main()试着访问矩阵之外的单元，get()函数返回了该矩阵的最后一个单元（已被设置为0）。该程序的输出结果如图16-8所示。

程序16-10　使用Matrix类作为方阵的容器

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
class Matrix {
  int *cells;                                  // heap array to house the matrix
  int size;                                    // number of rows and of columns
  int* make(int size)                          // private allocator function
  { int* p = new int [size * size];            // total number of elements
    if (p == NULL) { cout << "Matrix too big\n";  exit(0); }
    return p; }                                // return pointer to heap storage
public:
  Matrix (int sz) : size(sz)                   // conversion constructor
  { cells = make(size); }                      // heap memory is not initialized
  Matrix (const Matrix& m) : size(m.size)
  { cells = make(size); }                      // copy constructor: for safety
  Matrix& operator = (const Matrix& m);        // assignment operator
  int getSize() const                          // size of the side
  { return size; }
  int& get (int r, int c) const;               // access or modify a component
  ~Matrix() { delete [] cells; }               // destructor
  } ;

Matrix& Matrix::operator = (const Matrix& m  // assignment
{ if (this == &m) return *this;                // no work if self-assignment
  delete [ ] cells;                            // return existing memory
  cells = make(m.size);                        // allocate/set new memory
  size = m.size;                               // set the matrix size
  for (int i=0;  i<size*size;  i++)            // copy data
        cells[i] = m.cells[i];
  return *this; }                              // to support chain assignment

  int& Matrix::get (int r, int c) const
   {  if (r<0 || c<0 || r>=size || c>=size)  // check validity
        return cells[size*size-1];           // return last matrix cell
      return cells[r*size + c]; }            // return requested cell
void printMatrix(const Matrix& m)                  // client function
{ int size = m.getSize();
  for (int i=0; i < size; i++)                     // go over each row
  { for (int j=0;  j < size;  j++)                 // and each column
     cout <<setw(4) <<m.get(i,j);                  // print cell at m[i][j]
    cout << endl; }                                // end the current row
  cout << endl; }                                  // end the matrix

int main()
{ cout << endl << endl;
  int i, j, n = 5;  Matrix m1(n);                  // Matrix object
  for (i=0; i < n; i++)
    for (j=0;  j < n;  j++)                        // initialize cells
      m1.get(i,j) = (i+1) * (j+1);                 // m1[i][j] = (i+1)*(j+1);
  printMatrix(m1);                                 // print matrix state
  for (i=0; i < n; i++)                            // put 0's on main diagonal
    m1.get(i,i) = 0;                               // m1[i][i] = 0
  printMatrix(m1);                                 // print new state
  cout <<"m[10][10] = " <<m1.get(10,10) <<endl;    // out of range
  return 0;
  }
```

将get()函数变形为重载的函数调用运算符非常简单：用operator关键字加上运算符符号()来代替get即可。

```
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25

 0   2   3   4   5
 2   0   6   8  10
 3   6   0  12  15
 4   8  12   0  20
 5  10  15  20   0

m[10][10] = 0
```

图16-8　程序16-10的输出结果

```
int& Matrix::operator() (int r, int c) const
 {  if (r<0 || c<0 || r>=size || c>=size)          // check validity
      return cells[size*size-1];                   // return last matrix cell
    return cells[r*size + c]; }                    // return requested cell
```

这样可以使用函数调用语法来调用该函数，它是get()函数的同义词。

```
void printMatrix(const Matrix& m)              // client function
{ int size = m.getSize();
  for (int i=0; i < size; i++)                 // go over each row
  { for (int j=0;  j < size;  j++)             // and each column
      cout <<setw(4) <<m.operator()(i,j);      // cell at m[i][j]
    cout << endl; }                            // end the current row
  cout << endl; }                              // end the matrix
```

这段代码看起来有点怪，但它是正确的。在我们习以为常之前，所有的重载运算符都看起来有些奇怪。将函数调用语法变形为表达式语法也并不常见。C++规则的变形应用可能会产生像m()i,j这样的代码。相反，C++给以一些特殊的补充，可以将它写为m(i,j)。对一些C++程序员来说，这个语法看起来并不像是在访问矩阵的成员，但是对许多科学程序员来说，它与FORTRAN允许我们做的事情非常接近。

将程序16-10中调用get()函数的地方都换为调用重载的函数调用运算符operator()()后，即为程序16-11。（希望大家认为operator()()的名字是和其他任何函数名用一样的方式构成的。）其输出结果与程序16-10完全相同（见图16-8）。

程序16-11　使用重载函数调用运算符的Matrix类

```
#include <iostream>
#include <iomanip>
using namespace std;

class Matrix {
  int *cells;                                  // heap array to house the matrix
  int size;                                    // number of rows and of columns
  int* make(int size)                          // private allocator function
  { int* p = new int [size * size];            // total number of elements
    if (p == NULL) { cout << "Matrix too big\n";  exit(0); }
    return p; }                                // return pointer to heap storage
public:
```

```
  Matrix (int sz) : size(sz)                    // conversion constructor
  { cells = make(size); }                       // heap memory is not initialized
  Matrix (const Matrix& m) : size(m.size)
  { cells = make(size); }                       // copy constructor: for safety
  Matrix& operator = (const Matrix& m);         // assignment operator
  int getSize() const                           // size of the side
  { return size; }
  int& operator () (int r, int c) const;        // access or modify
  ~Matrix() { delete [] cells; }                // destructor
  } ;
Matrix& Matrix::operator = (const Matrix& m)      // assignment
{ if (this == &m) return *this;                   // no work if self-assignment
  delete [ ] cells;                               // return existing memory
  cells = make(m.size);                           // allocate/set new memory
  size = m.size;                                  // set the matrix size
  for (int i=0;  i<size*size;  i++)               // copy data
        cells[i] = m.cells[i];
  return *this; }                                 // to support chain assignment

int& Matrix::operator () (int r, int c) const
  {  if (r<0 || c<0 || r>=size || c>=size)        // check validity
        return cells[size*size-1];                // return last matrix cell
      return cells[r*size + c]; }                 // return requested cell

void printMatrix(const Matrix& m)                 // client function
{ int size = m.getSize();
  for (int i=0; i < size; i++)                    // go over each row
  { for (int j=0;  j < size;  j++)                // and each column
      cout << setw(4) << m(i,j);                  // print the cell
    cout << endl; }                               // end the current row
  cout << endl; }                                 // end the matrix

int main()
{ cout << endl << endl;
  int i, j, n = 5;  Matrix m1(n);                 // Matrix object
  for (i=0; i < n; i++)
    for (j=0;  j < n;  j++)                       // initialize cells
      m1(i,j) = (i+1) * (j+1);                    // m1[i][j] = (i+1)*(j+1);
  printMatrix(m1);                                // print matrix state
  for (i=0; i < n; i++)                           // put 0's on main diagonal
    m1(i,i) = 0;                                  // m1[i][i] = 0
  printMatrix(m1);                                // print new state
  cout << "m[10][10] = " << m1(10,10) << endl;    // out of bounds
  return 0;
  }
```

该设计不支持矩阵加法、乘法、比较及其他有用的操作。它的目的只是演示函数调用运算符的使用。

16.4 输入/输出运算符

C++标准库为所有的内部数据类型重载了输入运算符>>和输出运算符<<。显然，这些运算符并不支持程序员定义的类类型。因此，当需要输入或输出对象数据时，我们必须为对象的每个数据成员单独实施输入或输出操作。

为程序员定义的类重载输入/输出运算符是很好的。我们将这些操作封装在重载的运算符

中有助于简化客户代码的编写，消除客户代码对低级数据细节的管理，并将职责从客户代码推给服务器类。

16.4.1 重载运算符>>

例如，程序16-6中的String类动态管理其内存，并支持客户代码访问其内部数据。

```
class String {
  int size; // string size
  char *str;                                    // start of internal string
  void set(const char* s);                      // private string allocation
public:
  String (const char* s = "")                   // default and conversion
    { set(s); }
  String (const String &s)                      // copy constructor
    { set(s.str); }
  ~String() // destructor
  { delete [ ] str; }
  String& operator = (const String& s);         // assignment
  operator int() const;                         // current string length
  operator char* () const;                      // return pointer to start
} ;
```

可以为String类重载输入/输出运算符，使客户代码以下面的形式使用它：

```
int main ()
{ String s;
  cout << "Enter customer name:  ";
  cin >> s;                                     // accept name
  cout << "The customer name is: ";
  cout << s << endl;                            // display name
  return 0; }
```

重载输入运算符的界面应是什么形式呢？它是一个二元运算，对istream类型的对象（支持从库对象cin和磁盘文件中输入）和String类型的对象进行操作。String类的重载operator>>如下：

```
void String::operator >> (istream& in)
{ char name[80];                                // local storage for data
  in >> name;                                   // accept data
  delete [] str;                                // return existing memory
  set(name); }                                  // allocate/init new memory
```

请注意该函数是按引用传递参数，而不是按值传递，因为它要动态地管理其内存。注意该引用所指向的对象不是const对象，因为输入对象作为输入操作的结果被改变了。也请注意函数没有标注为const，因为它删除已存在的堆内存，并将指针指向另一堆内存区域，从而改变了对象数据成员的状态。

这都是可行的。但是这个运算符提供了一个糟糕的界面：根据C++将函数调用语法变形为表达式语法的规则，调用该函数时必须用一个String对象作为目标，一个istream对象作为参数。

```
s.operator >> (cin);              // equivalent to  s >> cin;
```

也可以像对待下标运算符和函数调用运算符那样，提供一个特别的补充方法，但是我们

不这样做。而且，如果cin对象不是左操作数，也不会有程序员愿意使用输入运算符。

既然这样不可行，我们可以把该运算符设计为istream类的成员。说比做要简单。istream类是一个库类，我们不能因为程序员定义的String类而摧毁它。

总之，我们不能将该运算符重载为istream类的成员函数，但可以（但是我们不想这样做）把它作为String类的成员来重载。那应该怎么做呢？我们舍弃所有其他方法，只能将该重载运算符定义为全局函数。

```
void operator >> (String& s, istream& in)      // global function
{ char name[80];                               // local storage for data
  in >> name;                                  // accept data
  String temp(name);                           // create/init new object
  s = temp; }                                  // copy it into the argument
```

这并不太好——不够简洁、而且有些慢，因为它首先创建了一个临时的String对象，然后才把它复制给参数。但是作为外部输入/输出的一部分，这样根本不会影响程序的性能。执行这个函数的时间远远少于用户响应的时间、或者从磁盘文件读入数据的时间。

下面，让我们看看如何使用函数调用语法来调用该函数：函数名、第一个参数、第二个参数。

```
operator >> (s, cin);                    // same as  s >> cin
```

这种形式不是我们想要的，问题在于String对象是第一个参数，而不是第二个。我们再次试着改变函数参数的次序。

```
void operator >> (istream& in, String& s)      // global function
{ char name[80];                               // local storage for data
  in >> name;                                  // accept data
  String temp(name);                           // create/init new object
  s = temp; }                                  // copy it into the argument
```

这样要好一些，String对象作为第二个参数这样的表达式语法才是我们所希望的。

```
operator >> (cin, s);              // same as  cin >> s;
```

下一步是将该函数定义为String类的友元，这样虽然它与目标对象无关，也仍能直接使用String类的非公共函数set()和数据成员。

```
class String {
  int size;                        // string size
  char *str;                       // start of internal string
  void set(const char* s);         // private string allocation
public:
friend void operator >> (istream& in, String& s);
. . . } ;                          // the rest of class String
```

这样就几近完美了。对于这个小例子来说，这的确很棒。但是，如果考虑到对待程序员定义类的方式要与对待内部类的方式大致相同，这个方法就不合适了。对于内部数据类型而言，iostream库支持链式运算：

```
double x, y;
cin >> x >> y;                   // same as cin >>x; cin >>y;
```

在我们的例子中，这样的客户代码无法工作，将产生语法错误。

```
String s;  int qty;
cout << "Enter customer name and quantity:  ";
cin >> s >> qty;                          // error: no chain calls
```

这段代码中最后一行语句表示什么意义呢？当然我们可以说C++允许这样做，因此应该可以编译，但是这种解释不够好。我们看一下（为所有类型重载的）函数operator>>的定义，其返回值为istream类型的引用。这是使得链式运算语法成为可能的惟一原因，没有使用其他特别的补充解释。

首先，我们用对象cin和s作为参数调用该运算符函数。当它返回一个istream对象的引用时，将该引用传送给另一版本的运算符函数（该版本来自于库类istream），该函数用qty变量作为参数。

```
(operator>>(cin,s)).operator>>(qty);       // same as cin >>s >>qty;
```

但我们定义的函数返回值是void类型，不是istream&，因而给它发送任何消息都是毫无意义的。补救措施很简单：重新定义该函数的返回类型为istream&。

程序16-12中实现了String类，并将operator>>重载为该类的友元。该运算符返回一个istream的引用以支持链式运算。程序的输出结果如图16-9所示。

程序16-12　为一个程序员定义的类型重载输入运算符

```
#include <iostream>
using namespace std;

class String {
  int size; // string size
  char *str;                                    // start of internal string
  void set(const char* s);                      // private string allocation
public:
friend istream& operator >> (istream& in, String& s);
  String (const char* s = "")                   // default and conversion
    { set(s); }
  String (const String &s)                      // copy constructor
    { set(s.str); }
  ~String() // destructor
  { delete [ ] str; }
  String& operator = (const String& s);         // assignment
  char* get () const                            // return pointer to start
  { return str; }
} ;
void String::set(const char* s)
   { size = strlen(s);                          // evaluate size
     str = new char[size + 1];                  // request heap memory
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s); }                           // copy client data to heap

String& String::operator = (const String& s)
{ if (this == &s) return *this;                 // no work if self-assignment
  delete [ ] str;                               // return existing memory
  set(s.str);                                   // allocate/set new memory
  return *this; }                               // to support chain assignment

istream& operator >> (istream& in, String& s)   // global friend
{ char name[80];                                // local storage for data
  in >> name;                                   // accept data
```

```
  delete [] s.str;                                    // return existing memory
  s.set(name);                                        // allocate/init new memory
  return cin; }                                       // important for chain work

int main ()
{
  String s;  int qty;                                 // local variables
  cout << "Enter customer name and quantity:  ";
  cin >> s >> qty;                                    // accept name, quantity
  cout << "The customer name is: ";
  cout << s.get() << endl;                            // using public methods
  cout << "Quantity ordered is:  ";
  cout << qty << endl;
  return 0;
  }
```

```
Enter customer name and quantity:  Simons  25
The customer name is: Simons
Quantity ordered is:  25
```

图16-9 程序16-12的输出结果

对待程序员定义类型的方式与对待C++中内部数据类型的方式应该相同，本例很好地支持了这个概念。

16.4.2 重载运算符<<

与operator>>类似，可以为程序员定义的类型重载输出运算符operator<<。

与重载输入运算符operator>>类似，不要将输出运算符作为某个程序员定义类型（例如String）的成员函数。否则会迫使我们使用糟糕的语法，即String对象在运算符的左边，而输出对象cout在运算符右边。

```
String s;
s << cout;                  // same as  s.operator << (cout);
```

与重载operator>>类似，也不能将输出运算符作为库输出流ostream类的成员函数。惟一可用的选择是作为全局函数来实现。要保证ostream对象作为第一个参数，而不是第二个。否则，我们会再次使用String对象出现在运算符的左边的语法。

```
void operator << (ostream& out, const String& s)
{ out << s.get(); }
```

这里没有必要将该函数定义为程序员定义类型的友元，因为它能访问所有需要的信息。而无论该函数是否访问数据，大多数程序员都喜欢让它成为友元。

该函数用于输出单个数据项，而不用于链式运算。

```
cout << "The customer name is: ";
cout << s;
cout << endl;
```

这样当然很不方便。与重载输入运算符类似，补救措施是返回对象的引用，这里是作为输出类ostream对象的引用。

```
ostream& operator << (ostream& out, const String& s)
{ return out << s.get(); }
```

现在程序员定义类型的输出链式运算也可行了，就像是内部数据类型那样。

```
cout << "The customer name is: " << s << endl;        // nice syntax
```

程序16-13中实现了String类及其重载的operator<<类友元。该函数返回一个ostream引用以支持链式运算。程序的输出结果如图16-10所示。

程序16-13　为程序员定义的类型重载输入/输出运算符

```
#include <iostream>
using namespace std;

class String {
  int size;                                    // string size
  char *str;                                   // start of internal string
  void set(const char* s);                     // private string allocation
public:
friend istream& operator >> (istream& in, String& s);
friend ostream& operator << (ostream& out, const String& s);
  String (const char* s = "")                  // default and conversion
    { set(s); }
  String (const String &s)                     // copy constructor
    { set(s.str); }
  ~String()                                    // destructor
  { delete [ ] str; }
  String& operator = (const String& s);        // assignment
  char* get () const                           // return pointer to start
  { return str; }
} ;

void String::set(const char* s)
    { size = strlen(s);                        // evaluate size
      str = new char[size + 1];                // request heap memory
      if (str == 0) { cout << "Out of memory\n"; exit(0); }
      strcpy(str,s); }                         // copy client data to heap

String& String::operator = (const String& s)
{ if (this == &s) return *this;                // no work if self-assignment
  delete [ ] str;                              // return existing memory
  set(s.str);                                  // allocate/set new memory
  return *this; }                              // to support chain assignment

istream& operator >> (istream& in, String& s)
{ char name[80];                               // local storage for data
  in >> name;                                  // accept data
  delete [] s.str;                             // return existing memory
  s.set(name);                                 // allocate/init new memory
  return cin; }
ostream& operator << (ostream& out, const String& s)
 { return out << s.str; }                      // it is allowed to a friend

int main ()
{ cout << endl << endl;
  String s;  int qty;                          // local data
  cout << "Enter customer name and quantity:  ";
```

```
cin >> s >> qty;                                    // accept name and quantity
cout << "The customer name is: " << s << endl;      // very nice
cout << "Quantity ordered is:  " << qty << endl;
return 0;
}
```

```
Enter customer name and quantity:  Smith  42
The customer name is: Smith
Quantity ordered is:  42
```

图16-10 程序16-13的输出结果

16.5 小结

本章的一系列讨论主要是围绕一个概念进行的：即编写函数使客户代码能以对待内部数据类型的方式对待程序员定义的类型。

我们讨论了一元运算符、前缀和后缀加1和减1运算符及转换运算符，它们在C++程序中经常使用。我们还讨论了转换构造函数和转换运算符，它们也削弱了C++的强类型规则，为对象的处理提供了更大的灵活性。

我们还回顾了下标和函数调用运算符，奇怪的是这两个运算符不遵循将函数调用语法转换为表达式语法的一般规则。与大多数C++运算符不同，只能将它们重载为成员函数，而不能重载为全局函数。这两个运算符使用得不多，但是，如果使用了它们就会给客户代码产生很好的影响。

最后，我们对输入/输出运算符进行了讨论，这是运算符重载真正有意义的论题。这两个运算符允许客户代码混用程序员定义类型和内部数据类型的对象。虽然从软件工程的角度来看并没有什么显著之处，但是它们简化了客户代码。

重载输入/输出运算符的应用非常广泛。希望大家能够自如地使用它们。

第17章 模板：另一个设计工具

本书的最后两章将介绍高级的C++程序设计概念：用模板和异常进行程序设计。

通常，容器类及其处理算法（排序、搜索等）是为某种特定的成员类型设计的。如果容器包含的是整数集合，则不能使用该容器存放如像账户这样的对象。如果一个函数是对一个整数数组进行排序，则不能使用该函数对库存项目进行排序，通常甚至不能使用它对双精度浮点数排序。C++模板允许程序员突破这种限制。使用模板，我们可以设计普通类（generic class）及其算法，然后指定某特定容器对象或函数调用所处理成员的类型。

使用异常处理机制可以简化实现复杂逻辑的代码。通常，处理算法使用C++中的`if`或`switch`语句将正常的数据处理与出错数据处理分开。对于多步骤的算法，主算法的源代码段与异常情况的源代码段分别写在同一源代码的不同分支里。这样常常使得源程序难于理解——因为主要的处理语句会迷失在大段的异常处理语句中。C++异常处理机制允许将程序中的异常处理代码隔离在其他遥远的源代码段中，从而简化基本的处理使程序更加容易理解。

模板和异常有一些共同特点：它们都比较复杂，增加了所写的应用程序的可执行代码的大小，并带来了额外的执行时间开销。

空间和时间开销是使用这些强大而复杂的程序设计方法的直接结果。如果要在严格的内存和执行速度限制下编写实时应用程序，我们可能不应该使用模板和异常处理机制。如果应用程序要在拥有大量内存和快速处理器的计算机上运行，空间和时间限制就不那么重要了。

逐渐将这些语言的特点引入我们的程序中，这可能是一个好的想法。如果这些方法只是很小程度地精简了我们的源代码，可能不值得那么麻烦地使用它们。程序设计中常见到这样的问题，应该从实践者自己的角度出发在优点和缺点之间进行权衡。务必使我们感兴趣的和追求的重要语言特征不会给维护人员带来太多的困难。

本章我们将介绍使用C++模板进行程序设计的方法。下一章将介绍C++的异常处理机制和其他一些在前面各章中没有讨论的高级语言特点。

17.1 类设计重用的一个简单例子

在程序员使用一种类型的对象代替另一种类型的对象时，C++的强类型规则会让编译程序指出程序设计错误。C++当然也允许有些例外：数值类型的值之间可以互换使用；若有转换构造函数或转换运算符，可以用程序员定义的类型代替其他类型；有继承关系的类之间也允许一些受限的替换。

但在类型值的使用上仍有许多限制。如果忽略操作值的类型，许多算法本质上是相同的。例如，在账户对象数组中搜索某个给定的账户时，将遍历数组的每个成员，并将拥有者姓名与所给定的姓名进行比较。类似地，在项目数组中搜索某个给定的库存项目时，将遍历数组的每个成员，并将项目的代号与所给定的代号进行比较。这些行为是相同的，但不能将库存项目数组作为参数传递给实现在账户数组中进行搜索的函数。我们必须编写另外一个函数。该函数和账户查找函数几乎相同。惟一区别是在比较操作时，一个函数将给定的姓名与账户

对象的拥有者的姓名作比较，另一个函数将给定的代号与项目对象中的代号进行比较。

栈、队列、表、树等容器类可以容纳不同种类的元素。复合类通常忽略元素的类型，而以相似的方式处理其元素。例如栈操作：将新的成员推到栈顶，从栈顶弹出一个成员，检查栈是否为空或者还有其他剩余元素等，它们都不依赖于栈中的元素特性。无论元素是字符、账户还是库存项目，都将以相同的方式处理。如果能够设计一个通用的栈，供给应用程序所需要的各种类型的元素使用，就太好了。但是C++强类型规则不允许这样做。一个包含字符的字符堆栈不能包含账户对象或者库存项目对象。而一个包含账户对象的账户堆栈不能包含字符或者库存项目。

下面讨论一个简单的例子：字符栈。这是一个常用的数据结构，它用于编译程序、计算器、屏幕管理及应用程序中其他需要支持后进先出（LIFO）协议的地方。在第8章，检查表达式括号的例子就使用了这种栈（称为临时存储器）作为底层数据结构。下面例子中的栈动态分配了堆上所需数目的字符，并支持push()、pop()和isEmpty()操作。弹出操作总是获取位于栈顶的符号，即最后一个压到栈中的符号。这样，当第一个符号被弹出后，下一个符号总是被推到栈顶。

```
class Stack{
  char *items;                    // stack of character symbols
  int top, size;                  // current top, total size
public:
  Stack(int);                     // conversion constructor
  void push(char);                // push on top of stack
  char pop();                     // pop the top symbol
  bool isEmpty() const;           // is stack empty?
  ~Stack();                       // return heap memory
};
```

程序17-1实现了Stack类及测试这个类的驱动程序。转换构造函数使用初始化列表初始化类的数据成员：栈所要求的字符数组的大小，数组中栈底的当前位置（即将插入下一个符号的位置下标）。该构造函数按客户代码所要求的大小分配堆内存。如果系统内存不足，则终止执行。

push()函数将其参数值插入到堆数组中。由于堆数组的大小是由客户代码指定的，客户代码应知道其算法需要多少栈存储器，当数组溢出时可以终止执行。但这样就将太多的任务推给了客户代码，客户代码同时还应该关注它自己的算法（例如检查括号是否匹配），不用考虑诊断信息的用户界面。因此将处理溢出问题的任务推给服务器类是更加合适的。

处理数组溢出的一个办法是终止程序运行。在服务器类而不是客户代码处理溢出的优点是使得客户代码变得简单，且不会包含与实现服务器细节有关的处理错误。因此，处理服务器问题（溢出）的另一个较好办法是在服务器而不是客户代码中处理。例如，在服务器对象中分配额外的内存，以防数组溢出。

分配多少额外的内存合适是有争议的。在程序17-1中，我们分配新的栈数组为当前数组大小的两倍，并将已有的栈内容拷贝到新分配的数组中，删除已存在的数组，然后使用这个是前一版本的两倍长的新分配的数组继续进行操作。客户代码完全不知道这些内存管理细节。

pop()函数很直观，它只是从栈中弹出栈顶字符，并修改指向栈顶的下标。对于一个大的数据结构来说，比较合适的做法是，观察顶部的位置，并在某时刻（例如现有数组的一半没有被使用时）返回现有的内存。本例很简单，没有必要这么做。

pop()函数可以检查栈是否为空。如果栈空了则发送一个消息（或返回值）。虽然该办法可行，但我们认为这会使Stack类和客户代码之间的通信变得不必要地复杂。如果在堆栈为空时客户试图弹出堆栈，它应该怎么做呢？在大多数情况下（见第8章的例子及程序8-10~程序8-13），空栈表明客户代码停止了某个处理而开始另一个处理。因此，没有必要让服务器类卷入应用程序的相关决定中。客户代码在每次调用pop()之前应调用栈方法isEmpty()，如果栈非空，则调用pop()方法，否则就做其他事情。在程序17-1中，我们终止了算法，因为空栈表示处理结束。

最后两个方法isEmpty()和Stack析构函数无足轻重。isEmtpy()检查栈下标是否已返回到初始位置。析构函数还回其生命期中分配给对象的堆内存。

Stack类对象只能存储指定类型的元素，而不能进行其他操作。这些对象之间不能相互赋值，也不能用其中一个对象初始化另一个对象。在形式上，可以对任意的C++变量（包括Stack对象）实施这些操作，但实际上却不支持在初始化或赋值时使用Stack对象。这意味着为Stack类增加一个拷贝构造函数和赋值运算符函数是多余的。另一方面，可以将这些函数原型定义为私有的。例如，当要按值传递一个Stack对象时会出现语法错误。

为了演示，程序17-1最初为Stack对象分配了一个很小的数组。因此，可以看到报告数组大小改变的调试信息。程序的输出结果如图17-1所示。

程序17-1　容纳字符的Stack类

```
#include <iostream>
using namespace std;

class Stack {
  char *items;                                  // stack of character symbols
  int top, size;                                // current top, total size
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                                   // conversion constructor
  void push(char);                              // push on top of stack
  char pop();                                   // pop the top symbol
  bool isEmpty() const;                         // is stack empty?
  ~Stack();                                     // return heap memory
} ;

Stack::Stack(int sz = 100) : size(sz),top(0)
{ items = new char[sz];                         // allocate heap memory
  if (items==0)
  { cout << "Out of memory\n";  exit(1); } }
void Stack::push (char c)
 { if (top < size)                                // normal case: push symbol
     items[top++] = c;
   else  // recover from stack overflow
     { char *p = new char[size*2];               // get more heap memory
       if (p == 0)                              // test for success
          { cout << "Out of memory\n";  exit(1); }
       for (int i=0;  i < size;  i++)           // copy existing stack
          p[i] = items[i];
       delete [] items;                         // return heap memory
       items = p;                               // hook up new memory
       size *= 2;                               // update stack size
```

```
        cout << "New size: " << size << endl;
      items[top++] = c; } }                          // push symbol on top

char Stack::pop()
{ return items[--top]; }                             // pop unconditionally

bool Stack::isEmpty() const                          // anything to pop?
{ return top == 0; }

Stack::~Stack()
{ delete [] items; }                                 // return heap memory

int main()
{
  char data[] = "abcdefghij";                        // pre-canned input data
  Stack s(4);                                        // Stack object
  int n = sizeof(data)/sizeof(char)-1;               // input data count
  cout << "Initial data:  ";
  for (int j = 0;  j < n;  j++)                      // print initial data
  { cout << data[j] << " "; }
  cout << endl;
  for (int i = 0;  i < n;  i++)                      // push data on the stack
    s.push(data[i]);
  cout << "Inversed data: ";
  while (!s.isEmpty())                               // pop until stack is empty
    cout << s.pop() << " ";
  cout << endl;
  return 0;
  }
```

```
Initial data: a b c d e f g h i j
New size: 8
New size: 16
Inversed data: j i h g f e d c b a
```

图17-1 程序17-1的输出结果

该设计有一个问题：当需要一个容纳其他类型元素的容器时，必须从头开始重复设计。上面元素类型的所有实例都得用其他元素类型的实例替换。例如，如果想要一个整数栈而不是字符栈，栈的定义应为如下形式：

```
class Stack {
  int *items;                        // stack of integer symbols
  int top, size;                     // current top, total size
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                        // conversion constructor
  void push(int);                    // push on top of stack
  int pop();                         // pop the top symbol
  bool isEmpty() const;              // is stack empty?
  ~Stack();                          // return heap memory
} ;
```

对于双精度浮点数值，该类必须再次修改。

```
class Stack {
```

```
    double *items;                    // stack of double symbols
    int top, size;                    // current top, total size
    Stack(const Stack&);
    operator = (const Stack&);
public:
    Stack(int);                       // conversion constructor
    void push(double);                // push on top of stack
    double pop();                     // pop the top symbol
    bool isEmpty() const;             // is stack empty?
    ~Stack();                         // return heap memory
};
```

注意，由一个全局的编辑器并不能胜任这个工作。为了满足设计要求，我们必须将指针定义、push()参数表及pop()返回值类型中的int类型改为double。数据成员top和size的定义不应该改变，构造函数的参数类型也不必改变。因此，这个设计的重用需要小心从事。这绝非是一件不费脑子的工作。

重用容器的另一个办法是使用一个通用的“参数”类型。该类型既不能为程序员定义类型，也不能为内部数据类型。例如，Stack类可按如下方式定义：

```
class Stack {
    Type *items;                      // stack of symbols of type Type
    int top, size;                    // current top, total size
    Stack(const Stack&);
    operator = (const Stack&);
public:
    Stack(int);                       // conversion constructor
    void push(Type);                  // push on top of stack
    Type pop();                       // pop the top symbol
    bool isEmpty() const;             // is stack empty?
    ~Stack();                         // return heap memory
};
```

这段代码只有当编译程序知道了Type的类型后才能编译，一旦定义了Type，就能编译该程序了。这样很好，因为它使得替换更为简单且不容易出错。我们应该只替换Type类型的实例，而且Type类型可用typedef来定义。例如：

```
typedef char Type;                    // type is equivalent to char
```

必须要让编译程序在处理Stack类定义之前发现该定义，编译程序将用char关键字替换Type标识符的每个实例，并编译其结果类。

有了这个方法后，类设计的重用就不会被随机的错误破坏了。当要为另一个类型的元素构造栈时，只需用另一类型的名字替换typedef语句中的char关键字就可以了。不会出现意外错误。程序17-2的Stack中Type类型为int，该程序的输出结果如图17-2所示。

程序17-2　容纳整数的Stack类设计的重用

```
#include <iostream>
using namespace std;

typedef int Type;                             // portable type definition

class Stack {
  Type *items;                                // stack of items of type Type
  int top, size;                              // current top, total size
```

```
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                                  // conversion constructor
  void push(const Type&);                      // push on top of stack
  Type pop();                                  // pop the top symbol
  bool isEmpty() const;                                // is stack empty?
  ~Stack();                                            // return heap memory
} ;

Stack::Stack(int sz = 100) : size(sz),top(0)
{ items = new Type[sz];                                // allocate heap memory
  if (items==0)
  { cout << "Out of memory\n";  exit(1); } }

void Stack::push (const Type& c)                       // pass by reference
 { if (top < size)                                     // normal case: push symbol
     items[top++] = c;
   else  // recover from stack overflow
     { Type *p = new Type[size*2];                     // get more heap memory
       if (p == 0)                                     // test for success
          { cout << "Out of memory\n";  exit(1); }
       for (int i=0;  i < size;  i++)                  // copy existing stack
          p[i] = items[i];
       delete [] items;                                // return heap memory
       items = p;                                      // hook up new memory
       size *= 2;                                      // update stack size
       cout << "New size: " << size << endl;
       items[top++] = c; } }                           // push symbol on top

Type Stack::pop()
{ return items[--top]; }                               // pop unconditionally

bool Stack::isEmpty() const                            // anything to pop?
{ return top == 0; }

Stack::~Stack()
{ delete [] items; }                                   // return heap memory

int main()
{
  Type data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } ;
  Stack s(4);                                          // stack object
  int n = sizeof(data)/sizeof(Type);                   // input data count
  cout << "Initial data:  ";
  for (int j = 0;  j < n;  j++)
   { cout << data[j] << " "; }                         // print input data
  cout << endl;
  for (int i = 0;  i < n;  i++)
   { s.push(data[i]); }                                // push data on the stack
  cout << "Inversed data: ";
  while (!s.isEmpty())                                 // pop until stack is empty
    cout << s.pop() << " ";
  cout << endl;
  return 0;
  }
```

```
Initial data: 1 2 3 4 5 6 7 8 9 0
New size: 8
New size: 16
Inversed data: 0 9 8 7 6 5 4 3 2 1
```

图17-2　程序17-2的输出结果

使用typedef方法可重用类的设计，该方法不仅适用于内部数据类型，也同样适用于程序员定义的任意类型，如账户、库存项目、矩形等。其前提条件是这些元素类型都支持容器类中元素对象执行的各种操作。这并不太困难，但是我们要确保在容器代码中识别出这些操作，因为它们通常是隐式的。

在Stack例子中，容器类在堆中创建一个元素数组，这意味着元素类必须要提供一个缺省的构造函数。这对于内部数据类型不成问题，但对于程序员定义类型可能成为问题。

在push()方法中，当一个元素对象插入到容器时要使用赋值。如果元素类不动态地处理其内存，则不会有问题。如果元素类要处理堆内存，它必须提供一个重载的赋值运算符。注意该容器的赋值运算符应定义为私有的——我们在这里讨论的是元素类的赋值运算符。

另一个要求元素类的设计能够提供重用的问题是参数传递和从容器类方法返回值。对于内部数据类型，该问题无关紧要。因此在程序17-1中，push()方法有一个值参，pop()方法返回这个值。在程序17-2中，push()方法将参数作为一个常量引用进行传递，以避免完整性问题及对性能的不良影响（详见第11章对这些问题的详细讨论）。为了与程序17-1相容，pop()方法仍返回值。但是许多容器设计者避免从容器方法返回值而按引用传递参数。

该方法允许我们在另一个程序中为支持赋值和拷贝的任意类型重用容器的设计。但如果在同一程序中使用了不同的栈，该方法就不起作用了。Type类型在编译时只能有一个意义。

在同一程序中为不同类型的元素使用同一容器时，我们只有求助于手工编辑方法了。每一个栈的名字都应互不相同，例如，charStack、intStack、pointStack等等，各个栈的代码和界面都应做相应的调整。

```
class doubleStack {
  double *items;                    // edit component type
  int top, size;                    // leave the type the same
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                       // leave the type the same
  void push(double);                // edit parameter type
  double pop();                     // edit return type
  bool isEmpty() const;
  ~Stack();
};
```

如果分别编辑每一个类的源代码，以后的修改就变得更加繁重，而且容易出错。不同的类名拥塞着项目的名字空间，这样有可能出现名字冲突。

使用宏可以自动地生成新的类名和代码，但这种重用方法复杂且容易出错。我们不认为今天的C++程序员还应该学习如何写宏，因为它是设计重用所弃用的方法。为了满足大家的好奇心，下面是这个栈的宏：

```
#define MakeName(a,b) a/**/b
```

```
#define DefineStack(Type)        \
class MakeName(Type,Stack) {     \
  Type *items;                    \
  int  top, size;                 \
Stack(const Stack&);              \
  operator = (const Stack&);      \
public:                           \
  Stack(int sz = 100) : size(sz),top(0) \
  { items = new Type[sz];                \
    if (items==0)                        \
      { cout << "Out of memory\n";  exit(1); } }   \
  void push(const Type& c)          \
  { if (top < size)                  \
     items[top++] = c;               \
    else                              \
     { Type *p = new Type[size*2];\
       if (p == 0)                    \
          { cout << "Out of memory\n";  exit(1); } \
       for (int i=0; i<size; i++) \
          p[i] = items[i];          \
       delete [] items;             \
       items = p;                    \
       size *= 2;                    \
       cout << "New size: " << size << endl; \
       items[top++] = c; } }        \
  Type pop()                        \
  { return items[-top]; }          \
  bool isEmpty() const              \
  { return top == 0; }              \
  ~Stack()                          \
  { delete [] items; }              \
} ;
```

客户必须首先用宏的开始处定义的名字DefineStack定义栈的类型。

```
DefineStack(int);
```

这就将（括号中指定的）类型与Stack（MakeName宏中的第二个参数）串接起来产生intStack。该语句同时也为整数栈定义了处理代码。然后客户代码就可以声明并使用一个合适的栈对象了。

```
intStack s(4);
```

因为预处理程序将所有的代码都转化为一行代码，所以调试起来很困难。宏中常常使用的词汇替换可能产生不正确的代码。因此，这并不是重用类设计的好方法。

17.2 模板类定义的语法

C++支持另一种重用类设计的方法，称为模板类（template class）。我们不再创建有固定类型元素的类，而是在创建类时将其元素的类型作为类的参数对待。

参数有程序员定义的名字，例如Type、T、Tp等。（对于任何参数，都是由程序员来决定如何称呼它们。）其实际参数可以为任意的类型：既可以是内部数据类型，也可以是程序员定义类型。当客户代码实例化该类的对象时，指定类中代替类参数使用的实际类型。

17.2.1 模板类说明

下面是栈的模板类说明，类型参数由程序员定义的名字Type表示。

```
template <class Type>                    // Type is given at instantiation
class Stack {
  Type *items;                           // actual type will be used
  int top, size;
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);
  void push(const Type&);                // actual type will be used
  Type pop();                            // actual type will be used
  bool isEmpty() const;
  ~Stack();
} ;
```

从理论上说，这是函数参数概念的延伸。在编写一个函数时，将给出每个参数名，这些参数名只是别名，将在稍后定义。函数指定了实际参数值上要执行的所有操作，但这些实际参数值在编写函数时是不知道的，只有在函数调用时才知道实际参数的值。在函数中，所有出现的形式参数的名字用实际参数的名字取代，然后在这个参数值上进行计算。

使用带参数的函数的优点是在设计算法时，不需要考虑用于计算操作的实际值，可对任意值实施计算操作，而且这些值只有在函数调用时才知道。如果在程序中其他地方需要同样的算法，只是参数值不同而已，这时就没有必要在源代码中再次实现该算法。同一函数可在（不需要修改的）程序的不同地方通过指定的实际参数名来调用。如果有必要，可使用同一参数再次调用同一函数。

类似地，模板告诉编译程序在要求使用某一特定类型的参数时如何生成代码。在编写模板时，给定了类型名，但该名字只是别名，将在稍后定义。模板指定了该类型上要执行的所有操作，但实际类型的名字在编写模板时是无法知道的，只有创建了该实际参数的对象时才知道实际类型的名字，并实施该类型值的计算操作。

使用模板类的好处是在设计算法时，不需要考虑用于计算操作的类型。可对任意类型实施计算操作（只要这些类型支持这些操作），该类型只有在对象实例化时才知道。如果程序中的其他地方需要同样的设计，只是类型不同而已，这时没有必要在源代码中再次实现它。同一模板类可在程序的不同地方为不同类型实例化，而不用进行任何修改。这些实例的惟一区别是实际类型的名字不同。如果有必要，可使用同一实际类型参数多次创建对象。

这样太好了。只需设计一个类，然后将该类作为模板（名字的来源地）使用以产生任意数目的特定类，这些特定类指定了使用的实际类型，用来代替我们在类设计中使用的参数类型。通过指定实际类型参数生成许多不同的类。模板类的其他术语有普通类和参数化类，它们可由不同实际类型的元素使用。

template是C++中的关键字。在类定义中，该关键字后有一对尖括号括起来的非空的参数列表。尖括号< >（不同于C++函数中的（ ））用来指定类型参数。参数列表中的每个模板类参数由class关键字和一个程序员定义的标识符组成。

尖括号中的每个模板参数是类型的一个占位符。一个参数化类（或普通类）可以有任意多个类型参数。尖括号中的多个模板参数之间用逗号隔开。

```
template <class T1, class T2, class T3>   // three type parameters
  class Triple;                           // class declaration
```

正如C++中的其他情况一样，参数列表中的class关键字与其他上下文中的class所代表的意义不一样。这里它用来表明在它后面的标识符是一个实际类型的占位符。这个实际类型不一定是类，可以是任意的内部数据类型，还可以是某一类型的表达式参数。

```
template <class Type, int size>           // a type and a value
  class Array;
```

这与函数参数类似，在Array类实例化时，客户代码必须提供指定类型的值（这里是int类型）。

17.2.2 模板实例化

与用C++创建任意类的对象相同，称创建一个模板类的对象为实例化。

当客户代码用模板类实例化某个特定的对象时，必须为每个模板参数提供实际类型参数。上面所描述的Stack模板不是一个类，它只是一个模板，不能用该模板直接创建Stack对象。没有指出实际类型的Stack对象十分荒谬可笑，就像push()没有指定应该压到栈中的实际参数值一样。

实际类型在模板实例化时作为尖括号中的类型名指定，尖括号紧接在模板类的名字后面。对象名的定义方式与非普通类的方式一样。如果有必要，可以照常使用构造函数参数。

```
Stack<int> is(50);          // stack of integers, length 50
Stack<char> cs(200);        // stack of characters, length 200
```

实例化与函数调用相似，函数的形式类型参数作为接受实际参数值的占位符。在函数定义中，用括号括起来的是形式参数表；在函数调用时，用括号括起来的是实际参数。在模板定义中，尖括号括起来的是模板参数；在模板对象实例化时，尖括号括起来的是实际类型。如果模板定义中不止一个类参数，模板实例化时必须在尖括号中指定相同个数的实际类型参数（以逗号隔开）。

与函数调用不同的是，在模板对象实例化中只使用实际编译时的值；这些值在客户代码中直接给出。C++并不能实现运行时类型变量：不能使用类型变量，也不能在模板类型参数中使用缺省的参数值。

当编译程序仔细分析模板类定义时，并不生成该类的目标代码，因为还不知道类元素的实际类型。因此编译程序维护其内部表及模板的有关信息，但不将目标代码增加到与模板定义源文件相应的目标文件中。

例如，当编译程序仔细解释Stack<int> is(50)的模板实例化时，将首先用实际模板类型参数替换模板形式参数，从而产生一个实际类定义（这里是Stack<int>）。这些代码不应包含到正生成的目标代码文件中，因为有可能在其他源文件中正在使用这个类。因此，编译程序将为Stack<int>生成另外一个目标文件，之后再创建Stack<int>类的对象。

```
template <class Type>
class Stack<int> {                  // type is given at instantiation
  int *items;                       // actual type is used
  int top, size;
  Stack(const Stack&);
  operator = (const Stack&);
```

```
public:
  Stack(int);
  void push(const int&);          // actual type is used
  int pop();                      // actual type is used
  bool isEmpty() const;
  ~Stack();
};
```

注意在定义普通模板时，push()函数的界面被定义为一个常数引用，因此，Stack<int>将参数定义为push(const int&)，这对于内部数据类型而言有点多余，但当它们表示比较大的复杂类时，可以提高实际类型的效率。然而，pop()函数返回的是一个值而不是引用，因此，客户代码不依赖于栈对象及其元素的生命周期。

Stack<int>类型的对象is的特点与其他任意的C++对象相同。对象本身没有说明是模板类的对象还是常规类的对象。使用非模板对象的地方也可以使用模板类对象，模板类对象也可作为参数传递，并能访问为普通类定义的消息。

```
if (!is.isEmpty())                // sending a message to object
   DebugPrint(is);                // passing object as parameter
```

使用非模板类名的地方也可以使用模板类名，但必须指定有实际类型名的参数列表。例如，这里的DebugPrint()函数应定义为接收某指定类型（这里是Stack<int>）参数的函数。

```
void DebugPrint(Stack<int>& stack);      // function prototype
```

17.2.3 模板函数的实现

在上面的例子中，DebugPrint()函数是一个常规的C++函数。该函数与其他我们在前面讨论过的C++函数的惟一区别是其参数是一个模板类对象。在该函数体中，参数栈被作为其类型已知的常规C++对象来处理。

如果不同类型的栈对象使用的调试算法相同，我们可能想在函数接口中进行定义。Stack<int>类型对于这样的函数而言太特殊化了：使用这种int类型后，函数只能接受这种类型的实际参数，而不能接受其他的栈类型的实际参数。为了支持通用性，应在该函数接口中说明任意类型的栈对象都是可接受的。为此，C++提供了模板（普通）函数的概念。

模板函数的定义方式与模板类相同：在template关键字后是尖括号括起来的类型参数表。参数表的每个实体都由两部分组成：关键字class、为类型名充当占位符的标识符。（再次强调，不必是类名，任意合法的C++类型都可以），模板参数表后是函数名和参数表所组成的函数头。在函数参数表中，使用了已实例化的模板类，指定了模板参数表的类型参数名而不是实际的类型。下面是普通函数DebugPrint()，它可以接受任意类型的栈参数。

```
template <class Type>                    // template parameter list
void DebugPrint(Stack<Type>& stack);     // function parameter list
```

在类作用域之外实现普通类的方法时使用同样的方法。这些函数是普通函数，只要将某个任意类型指定为实际类型即可适用于任何类型。（当然，函数可能希望实际类型的对象具有一些属性，例如，支持赋值运算符的能力。这依赖于函数实现的算法。）

```
void push(const Type&);      // no good outside of class braces
```

该函数原型存在两个问题：首先，它应该标明该函数属于Stack类；其次，应该说明标

识符Type是一个稍后将定义的类型参数，而不是已定义了的类型名。如果不这样做，编译程序将会认为Type没有定义。

C++中将模板类的成员函数视为模板函数来对待。一个模板函数的定义（或者声明）用template关键字开始，后面是尖括号括起来的模板参数表。参数表的每个元素都包括关键字class，其后是指定类型参数名的标识符。

```
template <class Type>                  // template parameter list
void Stack::push(const Type&);         // better but not good enough
```

现在，编译程序知道了标识符Type是一个类型参数名，将等待类对象实例化时的实际类型名。该函数定义的另一个问题是类名。我们使用了Stack，但程序中没有Stack类，因为Stack是一个模板类，而不是类。对于编译程序而言，没有任何修饰符的Stack名没有被定义。编译程序要知道这个Stack的元素类型。

我们应该告诉编译程序什么呢？在定义类时，我们也不知道Stack的类型将是什么，它几乎可以为任意类型。在实例化时会知道实际类型。我们所知道的只是这个类型将与函数参数中所指定的类型相同。因此，应在类型名后的尖括号中指定该类型。

```
template <class Type>                        // template parameter list
void Stack<Type>::push(const Type&);         // now it is good enough
```

在处理模板类时一定要注意这个重要的概念。函数原型的建立规则和其他任意的C++函数原型相同。要指定函数参数的类型（这里是Type），以及该函数所属的类（这里是Stack<Type>）。

在类定义范围外定义的每个成员函数都要使用模板参数表。

```
template <class Type>                                  // template parameter list
void Stack<Type>::push (const Type& c)                 // template prefix
 { if (top < size)                                     // normal case: push symbol
     items[top++] = c;
   else  // recover from stack overflow
     { Type *p = new Type[size*2];                     // actual type will be used
       if (p == 0)
          { cout << "Out of memory\n";  exit(1); }
       for (int i=0;  i < size;  i++)
          p[i] = items[i];                             // copy existing data
       delete [] items;
       items = p;                                      // hook up new heap array
       size *= 2;                                      // update stack size
       cout << "New size: " << size << endl;
       items[top++] = c; } }                           // push symbol on top
```

成员函数名中的作用域运算符应指定模板形式参数的标识符；普通类名后是由尖括号括起来的模板参数表。这与类定义范围外每次提到类名时要提供普通参数名的要求是一致的。

注意在作用域运算符的模板前缀中没有使用template或class关键字，而只有类型参数名。该模板前缀使得其后的函数可以使用形式参数，函数名本身不需要类型参数。

```
template <class Type>                                  // template parameter list
void Stack<Type>::push<Type> (const Type& c);          // overkill
```

这里的函数名push <Type>没有任何意义，只需要函数名本身就足够了。

类似地，在定义模板构造函数和析构函数时，只需要在模板前缀中而不是成员函数名中

定义一次模板参数。例如Stack<Type>::stack()，而不是Stack<Type>::stack<Type> ()。其中第二个stack是一个成员函数名而不是类型定义符。

```
template <class Type>
Stack<Type>::Stack(int sz = 100) : size(sz),top(0)
{ items = new Type[sz];
  if (items==0)
    { cout << "Out of memory\n";  exit(1); } }
```

对于析构函数也同样如此：~符号后是成员函数名而不是类名（即使两个名字是相同的），因此它不应该包含模板参数。在冒号作用域运算符之前是类名，因此应包含模板参数。

```
template <class Type>
Stack<Type>::~Stack()                    // special destructor syntax
 { delete [] items; }
```

注意析构函数在函数体内并未使用类型参数名，但仍必须在模板参数表和模板前缀中使用该类型参数。这是一般规则。函数定义中的关键字template及类型参数表必须包含该类的模板参数表中的所有类型参数。对于定义类名的模板前缀也同样如此。即使函数内并未使用全部的参数，仍必须列出所有的模板参数。

例如，成员函数isEmpty()对整数下标的值进行检查。无论在对象实例化时使用什么实际类型参数，其函数体保持不变，但是该成员函数的定义要求有完整的模板参数表和模板前缀。

```
template <class T>                       // it is not used in the function
bool Stack<T>::isEmpty() const           // return value of type bool
 { return top == 0; }                    // same body for any type
```

在这个定义中，我们使用了T标识符而不是Type表示类型参数。这当然是允许的，因为类型名只是一个占位符，只要保证在应该使用相同名字的地方使用相同的名字，我们就可以使用任何名字。在本例中，一致性要求是要求在同一函数的模板参数表中和模板前缀中使用相同的名字。对其他函数，可以使用另一个参数名。

程序17-3将栈实现为一个模板类。为了演示，我们使用了三种不同的参数名：Type、T和Tp。对于不同的成员函数，这些类型参数名是完全独立的。毕竟，这些函数可能在完全不同的源文件中实现。从软件工程的角度看，这不是一个好主意，因为C++类鼓励我们把属于一起的信息放在一起，但是语言的语法允许我们在不同的源文件中实现同一个类的不同成员函数。

在这个版本的程序中，客户代码实例化一个Point对象栈。Point类的对象有两个表示其坐标的整数域、一个空的缺省构造函数（支持数组的创建）和一个简单的拷贝构造函数（支持从pop()成员函数返回值）。对于这样一个简单的类而言，这两个构造函数都不是真正需要的。对于管理堆内存的类而言，这两个构造函数都是必要的。

为了与前面例子兼容，使用operator<<将Point坐标显示在屏幕上。该运算符重载为Point类的全局友元。程序的输出结果如图17-3所示。

程序17-3　容纳Point对象的Stack类设计的重用

```
#include <iostream>
using namespace std;
```

```
class Point {
       int x, y;
friend ostream& operator << (ostream& out, const Point& p);
public:
       Point() { }                                  // default constructor: empty
       Point(const Point &p)                        // copy constructor: for return
       { x = p.x;  y = p.y; }
       void set(int a, int b)                       // set Point coordinates
       { x = a;  y = b; }
} ;

ostream& operator << (ostream& out, const Point& p)
{ out << "(" << p.x << "," << p.y << ")";
  return out; }

template <class Type>
class Stack {
  Type *items;                                      // stack of items of type Type
  int top, size;                                    // current top, total size
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                                       // conversion constructor
  void push(const Type&);                           // push on top of stack
  Type pop();                                       // pop the top symbol
  bool isEmpty() const;                             // is stack empty?
  ~Stack();                                         // return heap memory
} ;

template <class Type>
Stack<Type>::Stack(int sz = 100) : size(sz),top(0)
{ items = new Type[sz];                             // allocate heap memory
  if (items==0)
  { cout << "Out of memory\n";  exit(1); } }

template <class T>
void Stack<T>::push (const T& c)
 { if (top < size)                                  // normal case: push symbol
     items[top++] = c;
   else  // recover from stack overflow
     { T *p = new T[size*2];                        // get more heap memory
       if (p == 0)                                  // test for success
          { cout << "Out of memory\n";  exit(1); }
       for (int i=0;  i < size;  i++)                  // copy existing stack
          p[i] = items[i];
       delete [] items;                                // return heap memory
       items = p;                                      // hook up new memory
       size *= 2;                                      // update stack size
       cout << "New size: " << size << endl;
       items[top++] = c; } }                           // push symbol on top

template <class Type>
Type Stack<Type>::pop()
{ return items[--top]; }                            // pop unconditionally

template <class Tp>
bool Stack<Tp>::isEmpty() const                     // anything to pop?
{ return top == 0; }
```

```
template <class Type>
Stack<Type>::~Stack()
{ delete [] items; }                                    // return heap memory

int main()
{
  Point data[5];
  data[0].set(1, 2); data[1].set(3, 4); data[2].set(5, 6);
  data[3].set(7, 8); data[4].set(9, 0);
  Stack<Point> s(4);                                    // stack object
  int n = sizeof(data)/sizeof(Point);                   // number of components
  cout << "Initial data:  ";
  for (int j = 0;  j < n;  j++)
    { cout << data[j] << " "; }                         // print input data
  cout << endl;
  for (int i = 0;  i < n;  i++)
    { s.push(data[i]); }                                // push data on the stack
  cout << "Inversed data: ";
  while (!s.isEmpty())                                  // pop until stack is empty
    cout << s.pop() << " ";
  cout << endl;
  return 0;
  }
```

```
Initial data: (1,2) (3,4) (5,6) (7,8) (9,0)
New size: 8
Inversed data: (9,0) (7,8) (5,6) (3,4) (1,2)
```

图17-3　程序17-3的输出结果

只要用指定了类型参数的值实例化类的对象，客户代码就可以使用其成员函数名。当成员函数作为消息发送给指定类的对象时，客户代码中的函数名不带前缀。

```
Stack<Point> s(4);                  // object is instantiated
. . .
for (int i = 0;  i < n;  i++)
  { s.push(data[i]); }              // no type specifiers here
```

这对于任何客户代码都成立，没有任何信息指明消息发送给模板类的对象而不是一般类的对象。在类定义中，情况就不一样了：类名后可能有也可能没有括号括起来的参数表。例如，对于实现容器的一个链表而言，一个结点类是一个模板，该模板有一个指向下一个结点的指针。因此，在定义结点类时必须使用这个类名来定义自己的数据成员。

```
template <class T>
struct Node {                       // public data
  T item;
  Node *next;                       // field next points to Node
  Node(const T&);                   // constructor
 } ;
```

在这里，假设编译程序知道next域与正被定义的类型相同，因而没有使用参数名。更为一致的办法是承认，只有定义了结点元素的类型时才有Node类。

```
template <class T>
struct Node {                   // public data
```

```
  T item;
  Node<T> *next;              // field next points to Node<T> *next
  Node(const T&);             // constructor
} ;
```

从软件工程角度而言，Node类的第二个版本比第一个版本要稍好一点。但对于编译程序来说，这两个版本都比较容易编译。

每个类的实例化将产生单独的类对象代码实例。根据编译程序的实现，目标代码将被放在一个单独的目标文件中，稍后再与其他目标代码文件连接。其后果之一是不再有源文件和目标文件的一一对应关系。这样开发人员就可能不太容易辨识其来源的对象文件。

模板实例化的增加可能会很大程度地增加编译和连接时间及目标代码的大小。有些编译程序可能会提供可行的办法控制这些不良影响。

当在类定义中以内联形式实现模板成员函数时，没有必要指定模板前缀和带有参数名的作用域运算符。

```
template <class Type>
class Stack {
  Type *items;                                  // stack of items of type Type
  int top, size;                                // current top, total size
  Stack(const Stack&);
  operator = (const Stack&);
public:

  Stack(int sz = 100) : size(sz),top(0)         // conversion constructor
  { items = new Type[sz];                       // allocate heap memory
    if (items==0)
       { cout << "Out of memory\n";  exit(1); } }

  void push(const Type& c)                      // push on top of stack
  { if (top < size)                             // normal case: push symbol
     items[top++] = c;
    else // recover from stack overflow
     { Type *p = new Type[size*2];              // get more heap memory
       if (p == 0)                              // test for success
          { cout << "Out of memory\n";  exit(1); }
       for (int i=0;  i < size;  i++)           // copy existing stack
          p[i] = items[i];
       delete [] items;                         // return heap memory
       items = p;                               // hook up new memory
       size *= 2;                               // update stack size
       cout << "New size: " << size << endl;
       items[top++] = c; } }                    // push symbol on top

  Type pop()                                    // pop the top symbol
  { return items[--top]; }                      // do it unconditionally

  bool isEmpty() const                          // is stack empty?
  { return top == 0; }

  ~Stack()                                      // return heap memory
  { delete [] items; }
} ;
```

当用typedef语句定义（见程序17-2）元素类型时，该类的定义看上去与一般类的定义

一样。

17.2.4 嵌套模板

一个模板类可使用其他的模板作为其数据成员。例如，一个元素为类T的栈模板Stack<T>可以拥有一个List<T>模板类型的数据成员。重要的是保证列表的元素与栈的元素的类型相同。栈模板的成员函数可把消息发送给列表模板对象以实现栈操作。

我们假设列表模板提供了以下操作：insert_as_first()将元素加到列表中作为列表的第一个元素，remove_first()删除列表的第一个元素。

```
template <class T>
class List {
    public:
      void insert_as_first(const T& x);
      T remove_first();
      bool empty();                    // is list empty?
   ...} ;                              // the rest of class List
```

使用一个列表模板作为栈的一个数据成员，从而实现了一个简单的栈。这里不需要实现动态内存管理，列表类负责这些管理。不需要构造函数或析构函数，也不必担心内存溢出。但仍需要关注栈的下溢，不过这不是栈设计人员的任务，客户代码应该构造栈处理算法以避免下溢。

```
template <class T>                 // same type T for Stack and List
class Stack {
  List<T> lst;                     // template data member
public:
  void push(const T&);             // const reference to T
  T pop();                         // return value of type T
  bool isEmpty(); } ;              // no need for a destructor
```

这样，栈成员函数的实现变得很简单。push()方法调用相应的列表函数，并依赖于列表函数的内存管理。

```
template <class T>
void Stack<T>::push(const T& item)
 { lst.insert_as_first(item); }              // push work down to list
```

类似地，pop()和isEmpty()栈成员函数将处理任务交给列表成员函数。

```
template <class T>
T Stack<T>::pop()                            // return value of type T
 { return lst.remove_first(); }              // push work down to list

template <class T>
bool Stack<T>::isEmpty()
 { return lst.empty(); }                     // call a similar function
```

当客户代码实例化Stack<int>对象时，成员函数将实例化为以下形式：

```
void Stack<int>::push(const int& item)
 { lst.insert_as_first(item); }

int Stack<int>::pop()                        // return value of type T
 { return lst.remove_first(); }
```

当客户代码声明一个栈对象实例，例如Stack<int>时，该实例化过程将迭代进行，编译程序生成List<int>的目标代码。同样，List<int>类的模板对象也可能需要其他模板。

如果不能实例化一个List<int>类的对象，栈实例化将出现一个实例化错误。可能是列表模板有问题，也有可能是模板参数的操作不合法（例如，可能没有定义类的比较操作，或者对一个内部数据类型参数应用了类的操作）。这就使调试模板类比调试一般的类要复杂得多。

17.3 多参数的模板类

在前面的例子中，我们使用的模板类只带有一个类型参数，尽管我们也曾提到一个模板类可以有几个类型参数。

这些参数可以与前面例子中的类型参数相似，甚至可与函数中一般的值参数相似。多个参数、混合类型参数及表达式参数为C++模板提供了更大的灵活性，但同时也增加了语法的复杂性。

17.3.1 多类型参数

我们考虑一个有多个类型参数的模板类。在类的数据成员、局部变量或方法参数中都要使用这些参数名。客户代码应在对象实例化时提供实际类型的名字。C++参数传递遵循位置原则：客户代码指定的第一个参数对应于模板类中的第一个参数，依次类推。

在模板类中指定类型参数表时，应为每个类型参数重复使用class关键字（参数间用逗号隔开）。这里是一个字典条目的模板类例子。该类有两个成员：关键字成员和信息内容成员。这两个成员可以是支持赋值运算和拷贝构造函数的任意类类型。

```
template <class Key, class Data>
class DictEntry {
    Key  key;                          // key field
    Data info;                         // information field
public:
  DictEntry () { }                     // empty default constructor
  DictEntry(const Key& k, const Data& d)
        : key(k),info(d) {}            // initialize data members
  Key getKey() const
  { return key; }                      // return key value
  Data getInfo() const
  { return info; }                     // return information value
  void setKey(const Key& k)
  { key = k; }                         // set key value
  void setInfo(const Data& d)
  { info = d; }                        // set information value
 } ;
```

这个设计相当幼稚，因为类的每个数据域的get()和set()函数完全可以不要，而将数据域定义为公共的，这样不会改变设计的质量。但是这并不重要，这个类的例子的目的是演示多个类型参数的使用。

该字典条目类实现了两个可以被成对处理的对象。例如，从客户代码的函数调用中可以返回该类的一个对象，这样比传递客户代码创建的对象指针或引用要简单快捷一些。当用关键字对象作为查找相关信息域值的下标时，可以比较方便地实现相关数组或者字典。为了使

搜索算法可行，key类除了支持赋值运算和拷贝构造函数外，还要支持比较操作。

这里没有必要提供DictEntry析构函数，因为类本身并不动态处理其内存（数据成员key和info）。如果任何成员类（Key或Data）要处理其他资源并拥有析构函数，在撤销字典条目对象过程中会自动调用该析构函数。这与拥有List数据成员的Stack类的上一个版本相似：List类有一个析构函数，但该版本的Stack类不需要析构函数。

要实例化一个DictEntry类的对象，客户代码必须定义两个实际类型：一个表示关键字域，另一个表示数据域。只要这些类型支持赋值运算和拷贝，我们可以以任何组合方式使用内部数据类型和程序员定义类。

```
DictEntry<Point,char*> entry;      // reference semantics for strings
```

这里，指向字符数组的字符指针可被程序中的其他对象共享。作为一个C++语言类型，该字符指针支持赋值和拷贝。当然，客户端代码程序员要注意：这里使用其引用语义，只是为了避免值语义的相关操作。除了支持共享之外，引用语义还可节省内存（不需要维护同一数据的多个副本）和执行时间（不需要从一个对象拷贝字符串到另一个对象）。

我们在第11章讨论的可怕问题源于析构函数强制撤销堆内存这个事实。既然这里没有将字符指针作为类实现，没有析构函数，因此，在这里就不存在第11章所描述的对程序完整性的威胁。

程序17-4演示了使用字典条目类的应用程序，该应用程序需要给屏幕上的点做注解，并查找每个给定点的注解。条目中的关键词域是Point类，该类与我们在程序17-3中使用的Point类相似（增加了一个简化数据初始化工作的一般构造函数和方便查找的比较运算符）。信息域用指向字符串的指针初始化。

在初始化字典条目数组之后，main()测试驱动程序打印数组中的每个条目。其执行结果如图17-4所示。

程序17-4　有两个类型参数的模板类的例子

```
#include <iostream>
using namespace std;

class Point {
      int x, y;
friend ostream& operator << (ostream& out, const Point& p);
public:
      Point() { }                                  // default constructor: empty
      Point(const Point &p)                        // copy constructor: for return
      { x = p.x;  y = p.y; }
      Point(int a, int b)                          // general constructor: set Point
      { x = a;  y = b; }
      void set(int a, int b)                       // set Point coordinates
      { x = a;  y = b; }
      bool operator == (const Point& p) const
      { return x == p.x && y == p.y; }
} ;

ostream& operator << (ostream& out, const Point& p)
{ out << "(" << p.x << "," << p.y << ")";
  return out; }

template <class Key, class Data>
```

```
class DictEntry {
    Key  key;
    Data info;
public:
  DictEntry () { }                                  // empty default constructor
  DictEntry(const Key& k, const Data& d)
       : key(k),info(d) {}                          // initialize data fields
  Key getKey() const
  { return key; }                                   // return key value
  Data getInfo() const
  { return info; }                                  // return information value
  void setKey(const Key& k)
  { key = k; }                                      // set key value
  void setInfo(const Data& d)
  { info = d; }                                     // set information value
 } ;

int main()
{
  DictEntry<Point,char*> data[5];
  data[0].setKey(Point(1,2)); data[0].setInfo("Initial stage");
  data[1].setKey(Point(3,4)); data[1].setInfo("Analysis");
  data[2].setKey(Point(5,6)); data[2].setInfo("Design");
  data[3].setKey(Point(7,8)); data[3].setInfo("Implementation");
  data[4].setKey(Point(9,0)); data[4].setInfo("Testing");
int n = sizeof(data)/sizeof(DictEntry<Point,char*>);     // risky
cout << "Associated Data:\n";
for (int j = 0;  j < n;  j++)
{ cout << data[j].getKey() << " "
       << data[j].getInfo() << endl; }                  // print input data
cout << endl;
return 0;
}
```

```
Associated Data:
(1,2) Initial stage
(3,4) Analysis
(5,6) Design
(7,8) Implementation
(9,0) Testing
```

图17-4 程序17-4的输出结果

该例子显示了带有多个类型参数的模板类的使用。使用多个参数可能引发名字冲突问题。那么，可以使用同一类名定义几个模板类吗？如果类只有一个类型参数，其答案显然是不能——当客户代码实例化一个类的对象时，编译程序不知道使用的是哪个类。

那么如果有多个参数呢？如果模板有不同数目的类型参数，是否可以使用同一类名定义这些模板类呢？

答案仍然是不行。模板类名不能重载。即使类型参数的个数不同，一个程序也只能用某给定的名字定义一个模板类。

17.3.2 带有常量表达式参数的模板

正如我们在前面提到过的，模板参数除了是类型外还可以是表达式，这些表达式可以是

任意的内部数据类型或者程序员定义类型。其类型在模板定义时显式地指定（不能作为类型参数），客户代码在实例化对象时提供该类型的值。

下面是一个Stack模板的例子。在这里，我们指定堆数组的初始大小为一个模板参数，而不是作为前面版本中的构造函数参数。

```
template <class Type, int sz>       // expression parameter
class Stack {
  Type *items;                      // stack of items of type Type
  int top, size;                    // current top, total size
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack();                          // default constructor
  void push(const Type&);           // push on top of stack
  Type pop();                       // pop the top symbol
  bool isEmpty() const;             // is stack empty?
  ~Stack();                         // return heap memory
} ;
```

当该类的对象实例化时，只为数组元素指定实际类型是不够的，还要指定数组的大小。

```
Stack<int,4> s;                   // stack object s
```

注意栈对象本身并没有任何参数，缺省构造函数不需要任何参数。还要注意参数必须按值传递——信息只能按一个方向流动，即从客户代码到模板对象。不允许使用引用参数，因为模板不能通过模板参数将信息传回给客户代码。我们只有出于这个目的才能使用普通的函数参数。

即使在参数定义中没有使用const修饰符，模板表达式参数本质上是常量。不能在模板函数中对它们进行修改，它们的实际值只能是常量表达式。不能使用非常数变量作为模板实例化的实际参数。在实例化时只能接受标注为const的字面值或标识符作为实际参数。

```
int size = 4;  const int length = 4;
Stack<int,size> s;                // syntax error: non-constant
Stack<int,length> s;              // OK: compile-time constant
```

在类定义作用域之外实现模板成员函数时，必须列出所有的模板参数。如果模板类有表达式参数，这些参数不仅要在模板参数表中还要在模板类前缀中列举出来。

```
template <class Type, int sz>       // expression parameter
Stack<Type,sz>::Stack()             // expression parameter
 : size(sz),top(0)
{ items = new Type[sz];             // use expression parameter
  if (items==0)
  { cout << "Out of memory\n";  exit(1); } }
```

与类型参数（和函数参数）相似，表达式参数也只是一个占位符，其名字并不重要，只要与函数中保持一致即可。模板类的不同函数之间不必保持一致的名字。下面是另一个栈函数，其中我们使用了不同的类型参数名和表达式参数名。

```
template <class T, int s>                    // different names for parameters
void Stack<T,s>::push (const T& c)           // consistent parameter names
 { if (top < size)                           // normal case: push symbol
     items[top++] = c;
   else
```

```
{ T *p = new T[size*2];                    // get more heap memory
  if (p == 0)
     { cout << "Out of memory\n";  exit(1); }
  for (int i=0;  i < size;  i++)           // copy existing stack
     p[i] = items[i];
  delete [] items;                         // return heap memory
  items = p;
  size *= 2;                               // update stack size
  cout << "New size: " << size << endl;
  items[top++] = c; } }                    // push symbol on top
```

与类型参数相似，每个成员函数的模板参数列表和类名前缀中都要将表达式参数列出来。即使函数体内不使用某表达式参数，也要将它列举出来。

```
template <class Type, int sz>
Type Stack<Type,sz>::pop()                 // parameters are not used
{ return items[--top]; }

template <class Tp, int s>
bool Stack<Tp,s>::isEmpty() const          // parameters are not used
{ return top == 0; }

template <class Type, int sz>
Stack<Type,sz>::~Stack()
{ delete [] items; }                       // parameters are not used
```

带有表达式参数的模板类的主要特点是每个实例化都表示不同的C++类型。这些不同的类型之间互不相容，在需要某种类型的对象时不能使用其他类型的对象代替。

```
Stack<int,4> s;              // stack object
Stack<int,8> s1;             // incompatible stack object
```

例如，我们考虑全局函数DebugPrint()，其参数为Stack<int,4>类。注意，该参数对象是按引用传递的，因为我们声明Stack<Type,sz>拷贝构造函数为私有函数，不允许按值传递栈对象。还要注意，该参数没有标注为const，因为它在函数执行过程中发生了改变（即使只是暂时改变）。

```
void DebugPrint(Stack<int,4>& s)           // no const modifier
{ Stack<int,4> temp;
  cout << "Debugging print: ";
  while (!s.isEmpty())                     // pop until stack is empty
  { int x = s.pop();  temp.push(x);        // save in temporary stack
    cout << x << " "; }                    // print each component
  cout << endl;
  while (!temp.isEmpty())                  // pop until stack is empty
  { s.push(temp.pop()); } }                // restore initial state
```

栈对象s和s1是两种不同类型的对象，对象s可作为参数传递给DebugPrint()。如果将对象s1作为参数传递给DebugPrint()，就会出现语法错误。

```
DebugPrint(s);               // OK
DebugPrint(s1);              // syntax error
```

计算出相同值的实际表达式是等价的。

```
const int length = 4;
Stack<int,length> s2;        // compatible with Stack<int,4>
```

就只带有类型参数的模板而言，所有使用同样的实际类型参数实例化的对象属于同一种类型，一个对象可代替另一个对象使用。下面，我们考察只带有一个类型参数的模板类。

```
template <class Type>
class Stack {
  Type *items;                          // stack of items of type Type
  int top, size;                        // current top, total size
  Stack(const Stack& = 100);
  operator = (const Stack&);
public:
  Stack(int);                           // conversion constructor
  void push(const Type&);               // push on top of stack
  Type pop();                           // pop the top symbol
  bool isEmpty() const;                 // is stack empty?
  ~Stack(); // return heap memory
} ;
```

这两个对象虽然有不同的初始数组长度，仍属于同一类型，可相互替代使用。

```
Stack<int> stack1(20);      // same type as other Stack<int> objects
Stack<int> stack2(50);
```

这样比带有表达式参数的模板类更为灵活和方便。通常，带有额外表达式参数且没有构造函数参数的模板类所能做的事，带有类型参数和构造函数参数的模板类都能做，而且对于后者来说，不同初始长度的对象是相容的。因而尽量避免使用带有表达式参数的模板，除非它对于只使用类参数的模板来说优势十分明显。

17.4　模板类实例之间的关系

模板类实例可作为实际类型参数实例化其他的模板类。例如，可使用下面的声明创建一个字典条目栈。

```
Stack<DictEntry<Point,char*> > stackOfEntries;          // 100 entries
```

注意，在两个“>”之间有一个空格。如果没有这个空格，编译程序将会误解该语句，并出现很多无关的错误信息，而其中没有任何一条信息指出这里需要一个空格。这是C++空格敏感的第二个地方。另一个空格也很重要的地方是，当没有使用参数名而为一个指针类型的函数参数定义缺省值时。

在上面的声明中，Stack实例化促使DictEntry实例化。一个优化的编译程序可能将目标代码缓存起来供以后编译重用。如果编译程序不这样做，编译和连接时间会显著增加。

不同实际类型（和表达式值）的模板实例化是分别进行的，它们之间没有任何关系，也不能相互访问。

例如，DictEntry<int,int>和DictEntry<float,record>是两个相互独立的不同类。Stack<int>和Stack<float>的实例化也是两个相互独立的不同类。不能用一种类的对象代替另一种类的对象使用。

一个类可以声明其所有的模板实例有一个共同的非模板基类：

```
template <class T>
class Stack : public BaseStack {
.... } ;
```

根据继承原则，Stack类的所有实例都可以访问BaseStack对象。这些实例不能互相访

问非公共成员。

17.4.1 作为友元的模板类

如果一个模板类的实例化对象的使用不依赖于其类型，一个非模板类（或函数）可以声明为这个模板类所有实例的友元。

```
template <class T>
 class Stack {
   friend class StackUser;
 .... ) ;
```

这里，无论使用什么类型作为实际类型，StackUser类都可访问Stack类的任意实例的非公共成员。

反过来，一个模板类（或函数）可以声明为一个非模板类的友元，即使其类型参数没有限制为任意实际值。

```
class Node {
template <class T> friend class Stack;
  int item;
  Node *next;                                  // Node *next' is also OK
  Node(const int val) : item(val)
   { next = NULL; }
 } ;
```

这里，Node类支持该信息域，并能链接到链表的下一个结点。除了使用构造函数初始化两个数据域外，它不需要其他任何成员函数。

Stack类的每个实例都是非模板类Node的友元，可以访问其非公共成员。如果抽取公共的代码可以减少目标代码的大小（和编译/连接时间），这样做就很有用。Stack类不在实例化（或数组溢出）时分配堆内存，而是每当将数据压到栈中时分配一个Node对象，当数据从栈中弹出来时回收栈顶的Node对象。

但这用处不是很大。一个（例如有整数信息的域）Node类型不能容纳客户代码想压进栈中的其他不同类型的对象。这意味着Node类必须也是一个模板。

这也意味着应该将Node类定义为模板类。这样就可以实例化不同类型的Stack对象，并用不同类型的item域访问不同类型的Node对象。

```
template <class Type>                          // template class
 class Node {
     friend class Stack<T>;                    // any type of component
  Type item;
  Node<Type> *next;                            // Node *next'  is also OK
  Node(const Type& val) : item(val)
   { next = NULL; }
 } ;
```

这种模板使用方法的科技术语是*无约束类型*（unbounded type）。参数Type不依赖于参数T，每个参数可独立地接受任意类型的实际类型值。考虑一下，这大大超过了我们的需求。这里，Stack类的每个实例（例如float类型）可以访问Node类的每个实例（例如Point类）的细节。该程序代码没有实现现实世界的真实模型。

我们需要在相关实例间建立一对一的映射关系。这样，一个整数栈只成为一个整数结点

的友元，而不是拥有其他类型的item域的结点的友元。为了实现这种映射，我们可以将一个友元（客户）模板类（这里是Stack）绑定到使用相同类型并提供服务的模板类（这里是Node）上。

```
template <class Type>                          // template class
 class Node {
     friend class Stack<Type>;                 // same type of component
  Type item;
  Node<Type> *next;                            // Node *next'  is also OK
  Node(const Type& val) : item(val)
    { next = NULL; }
 } ;
```

这里，对于Node的每个实例化所指定的类型（例如Point类），实例化为相同类型（Point类）的Stack作为该Node类实例的一个友元。

现在，Stack类有一个Node类指针的数据成员，该数据成员在Stack构造函数中初始化为0。当下一个结点压到栈中时，该指针指向这个新结点（新结点指向列表中的第一个结点）。成员函数isEmpty()检查该指针是NULL还是指向一个结点。这意味着当最后一个结点从栈中删除时，函数pop()应将该指针设为NULL。

```
template <class T>
class Stack {
  Node<T> *top;                        // Node *top;  is illegal here
public:
  Stack()                              // default: no initial length
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }              // does top point to a node?
  ~Stack();
 } ;
```

对于任意模板，在Node定义之外使用Node时必须用参数表限制。因此，Stack数据成员top不是Node*类型，而是Node<T>*类型，其中T为Stack类型参数。正是由于在Stack定义中进行这样的限制，一个Stack类对象的实例化将导致同一类型的Node类数据域的自动实例化。

Stack方法push()在堆中分配一个新的Node对象，调用Node构造函数初始化该Node对象的item为push()的参数值，新Node对象的next域设置为指向Stack域top当前正指向的结点，并将top域重置为指向新的Node对象。

```
template <class T>
void Stack<T>::push (const T& val)
 { Node<T> *p = new Node<T>(val);          // type Node<T>, not Node
   if (p == NULL)
    { cout << "Out of memory\n";  exit(1); }
   p->next = top;                          // point it to first node
   top = p; }                              // point to new node
```

没有必要在push()中测试数组溢出，因为在该实现中没有使用数组。但仍要测试Node对象的分配是否成功。注意指针的类型——不是Node*，而是Node<T>*。类似地，当new操

作要求堆空间时，其类型是Node<T>，而不是Node类型。在Stack实例化时要提供类型T。

Stack方法pop()将（Node<T>类型而不是Node类型的）局部指针指向栈的第一个结点，并将信息域拷贝到（T类型的）局部变量中，移动top域指向第二个结点，删除栈顶结点，因为再也不需要它了。

```
template <class T>
T Stack<T>::pop()                          // return value of type T
 { Node<T> *p = top;                       // Node of type T, not Node
   T val = top->item;                      // get the value of type T
   top = top->next;                        // point to the second node
   delete p;                               // return top node to heap
   return val; }
```

当pop()删除了列表中的最后一个结点时（而且域top没有指向第二个结点），top指针变为NULL. 为什么？因为当第一个结点在push()中插入时，p->next=top;语句将该域设为NULL（因为Stack析构函数把域top设置为NULL）。要确保同一类的成员函数之间通过类数据紧密耦合度。成员函数的源代码之间必须协调，以确保函数之间的正确协作。

Stack析构函数必须扫描剩余结点的链表，并将它们返回给堆。再次使用了带有T类型成员的Node<T>类型的局部指针p，将该指针指向表的第一个结点。注意，指向表中第一个结点的指针的数据成员top也和指针p:Node<T>属于同一类型。在while循环中，指针top指向下一个结点，删除指针p所指向的结点，并且指针p移到指向下一个结点。

```
template <class T>
Stack<T>::~Stack()
 { Node<T> *p = top;                       // type Node of type T
   while (top != NULL)                     // top is 0 when no nodes
     { top = top->next;                    // point to the next node
       delete p;                           // delete the previous node
       p = top; } }                        // traverse to the next node
```

这种方法的优点是Node类与Stack类是相互独立的。这样Node类可由其他的友元类使用，例如Queue、List等。由于该设计中所有Node成员（包括构造函数）都是私有的，非友元客户不能创建或访问Node对象。

另一种方法是为每个客户提供各自的私有服务器类。如果Node定义嵌套在客户的私有部分，则只有其客户（及其友元）才能访问Node类。这也将引发模板定义中的协作（映射）问题。

17.4.2 嵌套模板类

使用嵌套设计，可以将模板类Node的定义嵌套在处理Node对象的容器类定义中。由于Node定义完全处于容器类（如Stack）的作用域内，Node名字对于其他潜在客户（如Queue、List）是不可见的。因此，Node成员可定义为public，并且没有必要声明其单个客户（如Stack）为Node类的友元。

这是使用嵌套类设计的第一个尝试。使用关键字struct定义Node类，其所有的成员都是公共的。

```
template <class T>
class Stack {
```

```
  template <class Type>              // Is it legal? Is it needed?
    struct Node {
      Type item;
      Node<Type> *next;              // type depends on instantiation
      Node(const Type& val) : item(val)
      { next = NULL; } } ;
  Node<T> *top;                      // Stack data member
public:
  Stack() // default: no initial length
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }            // does top point to a node?
  ~Stack();
 } ;
```

该定义中有两个问题：首先，有些编译程序不接受嵌套的模板定义，它们只能处理全局模板。其次，没有必要使用无约束的模板类型。在这个设计中，Stack类和Node之间的映射是一对多：对于Stack类的任意类型参数，Node类可使用任意其他类型。Stack和Node之间应是一对一映射，而不是一对多：Stack类需要将一个Node对象实例化为与其实际类型相同的类型。

一个有效办法是将Stack类定义为模板，然后将Node类定义为一般的非模板类，它使用Stack类型参数定义其数据成员和其方法参数类型。

```
template <class T>
class Stack {
    struct Node {                                  // it depends on parameter type
        T item;                                    // same type as in Stack
        Node *next;                                // Node<T> is incorrect here
        Node(const T& val) : item(val)             // same type as in Stack
         { next = NULL; } } ;
   Node *top;                                      // it is not a template now
public:
  Stack()                                          // default: no initial length
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }                          // does top point to a node?
  ~Stack();
 } ;
```

Stack模板的每个实例都生成一个Node类，该Node类使用与Stack的实际类型参数相同的类型。因而没有必要在Stack定义内限制Node类型。

对于Stack成员函数也是如此。在成员函数中定义一个局部指针时，该指针定义为指向Node类型的指针，而不是指向Node<T>类型的指针。例如，这里的push()方法与前一版本几乎完全相同，只是指针p的定义不同。我们将前面版本中的指针定义加以注释，以便进行两个版本的比较。

```
template <class T>
void Stack<T>::push (const T& val)
// { Node<T> *p = new Node<T>(val);        // type Node<T>, not Node
 { Node *p = new Node(val);                // type Node, not Node<T>
   if (p == NULL)
```

```
    { cout << "Out of memory\n";  exit(1); }
   p->next = top;                          // point it to first node
   top = p; }                              // point to new node
```

容器类的其他方法也同样如此。程序17-5实现了有嵌套类Node的模板类Stack。程序的输出结果如图17-5所示。

程序17-5 带嵌套服务器类的模板类的例子

```
#include <iostream>
using namespace std;

template <class T>
class Stack {
    struct Node {                              // it depends on parameter type
        T item;                                // same type as in Stack
        Node *next;                            // Node<T> is incorrect here
        Node(const T& val) : item(val)         // same type as in Stack
         { next = NULL; } } ;
  Node *top;                                   // it is not a template now
public:
  Stack()// default: no initial length
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }                      // does top point to a node?
  ~Stack();
 } ;

template <class T>
void Stack<T>::push (const T& val)
// { Node<T> *p = new Node<T>(val);            // type Node<T>, not Node
 { Node *p = new Node(val);                    // type Node, not Node<T>
   if (p == NULL)
    { cout << "Out of memory\n";  exit(1); }
   p->next = top;                              // point it to first node
   top = p; }                                  // point it to new node

template <class T>
T Stack<T>::pop()                              // return value of type T
// { Node<T> *p = top;                         // type Node<T>, not Node
 { Node *p = top;                              // type Node, not Node<T>
   T val = top->item;                          // get the value of type T
   top = top->next;                            // point to the second node
   delete p;                                   // return top node to heap
   return val; }

template <class T>
Stack<T>::~Stack()
// { Node<T> *p = top;                         // type Node of type T
 { Node *p = top;                              // type Node of type T
   while (top != NULL)                     // top is 0 when no nodes
     { top = top->next;                    // point to the next node
       delete p;                           // delete the previous node
       p = top; } }                        // traverse to the next node
```

```
int main()
{
  int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } ;
  Stack<int> s;                              // stack object
  int n = sizeof(data)/sizeof(int);          // number of components
  cout << "Initial data:  ";
  for (int j = 0;  j < n;  j++)
    { cout << data[j] << " "; }              // print input data
  cout << endl;
  for (int i = 0;  i < n;  i++)
    { s.push(data[i]); }                     // push data on the stack
  cout << "Inversed data: ";
  while (!s.isEmpty())                       // pop until stack is empty
    cout << s.pop() << " ";
  cout << endl;
  return 0;
  }
```

```
Initial data: 1 2 3 4 5 6 7 8 9 0
Inversed data: 0 9 8 7 6 5 4 3 2 1
```

图17-5　程序17-5的输出结果

如上所示，对相互协调的模板类成员类型的绑定依赖于总体设计方法。令人沮丧的是适用于一种设计（如全局类）的方法不一定适用于其他的设计（如嵌套类）。无论如何应该确保：当将一个类实例化为某特定类型时，另一个类也要实例化为相同类型。

17.4.3　带静态成员的模板

如果一个模板类声明了静态数据，那么每个模板实例都将各自拥有这些静态成员集合。属于某特定实例的所有对象将共享相同的静态成员，但它们不能访问属于不同实际类型参数实例的静态成员。

例如，Stack类可声明其top数据成员为静态的。这是一种有趣的设计方法，item和next数据域移到Stack类中作为非静态数据成员，这时Node类还剩什么呢？什么都不剩了，它变得多余了。因此，该设计中可以将Node类去掉。

该设计中的Stack类结合了前面例子中的Stack类功能（调用push()、pop()、isEmpty()成员函数）及Node类的功能（item和next域）。因此，它有两个构造函数：缺省构造函数和转换构造函数。

缺省构造函数在客户代码中实例化Stack对象时调用，它不需要完成任何工作，但是必须消除语法错误。转换构造函数是在push()方法中调用的：当分配一个新的结点时，push()创建一个新的Stack对象而不是新的Node对象。该构造函数初始化item域（为存储的值）和next域（指向栈顶的结点）。

pop()函数使用局部指针删除栈顶结点，指针的类型是指向Stack<T>的指针。由于它是一个指向Stack类型对象的指针，就调用了Stack析构函数（代替了以前版本中的Node析构函数）。在以前的版本中，Stack析构函数删除了剩余的栈结点。而在这里，这是有害的，因为Stack类没有析构函数（它有什么也不做的缺省析构函数）。

```
template <class T>  class Stack {
 static Stack  *top;                          // static data member
 T item;  // from Node
 Stack *next;                                 // from Node
public:
  Stack() { }                                 // create object in client
  Stack(const T& val)
         : item(val), next(top)               // create new node in push()
    { top = this; }
  void push(const T& val)
    { Stack<T> *p=new Stack<T>(val); }         // no Node<T>, no Node
  T pop()
    { Stack<T> *p = top;
      T val = top->item;                  // no Node<T>, no Node
      top = top->next;                    // point to second node
      delete p;                           // delete top node: destructor
      return val; }
  int isEmpty() const
  { return top == NULL; }
  void remove()                           // no call to destructor
  { Stack<T> *p = top;                    // trailing pointer
    while (top != NULL)
    { top = top->next;                    // go to next node
      delete p;                           // delete previous node
      p = top; } }                        // catch up with next node
} ;
```

没有析构函数将有可能导致内存泄漏。为了避免内存泄漏，Stack类提供了remove()方法，该方法与前面版本中的Stack析构函数功能相同。此设计的不足是客户代码必须显式调用remove()方法处理栈中剩余的结点（如果有的话）。

对模板类的静态成员的初始化，不是（像通常类中的静态成员的初始化那样）在程序开始执行时进行，而是在实例化一个模板对象时进行。因为只有在这时，才创建了这个特定实际类型的静态成员。

初始化语句（在头文件中）的语法应包括以下方面：

- 指明静态成员属于模板。
- 指定静态成员的类型。
- 指定静态成员的作用域。
- 指定静态成员名和初始值。

下面是Stack的静态成员top的初始化：其类型是Stack<T>*，作用域是Stack<T>，名字为top，初始值为NULL。

```
template <class T>                          // it belongs to template
  Stack<T>*  Stack<T>::top = NULL;
```

客户只能声明某给定类型的一个对象。例如，对于整数栈，模板实例化如下所示：

```
Stack<int> s;                               // only one object per type
```

由于所有的整数栈都共享相同的指向链表头部的静态成员，因此最好不要实例化该类型的多个对象。

17.5 模板的规则说明

在C++中，模板的基础是假设不同数据类型的算法完全相同，这样只编写一个类而不用为每个类型编写单独的类，这是有意义的。但有时该假设不成立。算法对不同的数据类型是以相同的方式处理的，但是对于某些类型，算法的某些细节可能需要不同的实现。

例如，模板类Array包含了一组（某个元素类型的）数据，并允许客户代码检查（这种元素类型的）某给定元素是否在这组数据中。

```
template <class T>
class Array {
  T *data;                                    // heap array of data
  int size;                                   // size of the array
  Array(const Array&);
  operator = (const Array&);
public:
  Array(T items[], int n) : size(n)           // conversion constructor
  { data = new T[n];                          // allocate heap memory
    if (data==0)
       { cout << "Out of memory\n";  exit(1); }
    for (int i=0;  i < n;  i++)               // copy input data
      data[i] = items[i]; }
  int find (const T& val) const
  { for (int i = 0;  i < size;  i++)
    if (val == data[i])  return i;            // return index if found
    return -1; }                              // otherwise return -1
  ~Array()
  { delete [] data; }
 } ;
```

该模板类只有一个构造函数、一个find()方法和一个析构函数。构造函数为输入数据分配足够的堆内存，并将输入数组拷贝到堆内存中。find()方法对堆数组进行查找，如果没有找到参数值则返回-1；如果找到了则返回该值在数组中的下标。析构函数还回堆内存。

客户代码实例化int类型的Array对象，并将该对象初始化，打印出查找某特定值的结果。

```
int main()
{ int data1[] = { 1, 2, 3, 4, 5 } ;
  int n1 = sizeof(data1)/sizeof(int);       // number of components
  cout << "Initial data:  ";
  for (int j = 0;  j < n1;  j++)
  { cout << data1[j] << " "; }               // print input data
  cout << endl;
  Array<int> a1(data1,n1);                  // array object
  int item1 = 3;  int idx;
  if ((idx = a1.find(item1)) != -1)
    cout << "Item " << item1 <<" is at index " << idx << endl;
  return 0; }
```

这段代码对整数、字符甚至Point对象都是以相同的方式运行：对于每一种类型，Array对象都将包含输入值的独立副本；find()方法中的比较操作对于这些类型也都能正常工作。但如果Array对象被实例化为字符数组类型的成员，构造函数和find()方法都将有问题。

```
Array<char*> a2(data2,n2);
```

这里，数组data2[]是一个字符串数组。Array模板的构造函数将拷贝指向字符串的指针，而不是字符串本身。如果数据是程序中给定的（如上面的程序段），则不会有问题。但在实际生活中，数据来自于一个外部源（而不是硬编码的数组），每个输入值都必须分配独立的空间。Array构造函数没有这样处理：拷贝给容器的指针将指向客户空间的同一字符数组。与之类似，find()方法会比较字符串地址而不是字符串的内容。我们看到，对于Array成员的字符数组，模板类的一般形式不起作用——它需要在构造函数中拷贝字符串，在find()方法中比较字符串。

C++支持特化（specialization）的概念，以处理需要特殊对待的类型参数。对于每个特殊的类，应该提供独立的特化模板类。描述这种特化的语法是模板类本身的语法（带有模板参数表）及客户中的模板初始化语法（带有实际类型参数表）的综合。我们从模板参数列表中将类型参数移到实际类型列表中。如果模板参数列表括号中变空了也没关系。例如：Array模板类的头部：

```
template <class T>                  // remove class T from brackets
  class Array {                     // append <char*> to class name
```

变成

```
template <>                         // empty template parameter list
  class Array<char*> {              // actual type list
```

在这个模板特化的方法中，描述了对该特定类型要进行的处理。注意既要有模板类定义，也要有特化的模板类定义。只包括特化的模板类定义而没有模板本身的定义是不正确的。用实例化模板类对象的语法对特化的模板实例化：即使实际参数是在模板定义中指定的，也要在客户代码的类型名字中重复出现类型名。

```
Array<char*> a1(data2,n2);          // specialized template object
```

程序17-6中演示了完整的程序，包括模板Array及其被字符数组类型成员特化的模板。测试驱动程序初始化模板类对象a1和特化模板对象a2，并把消息发送给每个对象。程序的输出结果如图17-6所示。

程序17-6 模板类特化的例子

```
#include <iostream>
using namespace std;

template <class T>
class Array {
  T *data;                                          // heap array of data
  int size;                                         // size of the array
  Array(const Array&);
  operator = (const Array&);
public:
  Array(T items[], int n) : size(n)                 // conversion constructor
  { data = new T[n];                                // allocate heap memory
    if (data==0)
       { cout << "Out of memory\n";  exit(1); }
    for (int i=0;  i < n;  i++)
      data[i] = items[i]; }
  int find (const T& val) const
  { for (int i = 0;  i < size;  i++)
```

```
        if (val == data[i])
          return i;
      return -1; }
    ~Array()
    { delete [] data; }
  } ;

template <>                                          // empty template list
class Array <char *> {                               // type of specialization
  char* *data;                                       // heap array of data
  int size;                                          // size of the array
  Array(const Array&);
  operator = (const Array&);
public:
  Array(char* items[], int n) : size(n)              // conversion
  { data = new char*[n];                             // allocate heap memory
    if (data==0)
      { cout << "Out of memory\n";  exit(1); }
    for (int i=0;  i < n;  i++)
      { int len = strlen(items[i]);                  // special for strings only
        data[i] = new char[len+1];
      strcpy(data[i],items[i]); } }
  int find (const char*& val) const
  { for (int i = 0;  i < size;  i++)
      if (strcmp(val,data[i])==0)                    // special for strings only
        return i;
    return -1; }
  ~Array()
  { delete [] data; }
 } ;

int main()
{
  int data1[] = { 1, 2, 3, 4, 5 } ;
  char* data2[] = { "one", "two", "three", "four", "five" } ;
  int n1 = sizeof(data1)/sizeof(int);                // number of components
  int n2 = sizeof(data2)/sizeof(char*);
  cout << "Initial data:  ";
  for (int j = 0;  j < n1;  j++)
  { cout << data1[j] << " "; }                       // print input data
  cout << endl;
  for (int i = 0;  i < n2;  i++)
  { cout << data2[i] << " "; }                       // print input data
  cout << endl;
  Array<int> a1(data1,n1);                           // array object
  Array<char*> a2(data2,n2);                         // specialized object
  int item1 = 3;  int idx;
  char* item2 = "three";
  if ((idx = a1.find(item1)) != -1)
      cout << "Item " << item1 <<" is at index " << idx << endl;
  if ((idx = a2.find(item2)) != -1)
      cout << "Item " << item2 <<" is at index " << idx << endl;
  return 0;
  }
```

C++还支持部分特化即可以只为几个类型参数中的一个进行特殊处理。例如，程序17-4中的模板类DictEntry有两种类型参数：

```
Initial data: 1 2 3 4 5
one two three four five
Item 3 is at index 2
Item three is at index 2
```

图17-6 程序17-6的输出结果

```
template <class Key, class Data>
class DictEntry {
   Key  key;
   Data info;
public:
...} ;                                  // the rest of the class
```

对于实例化为字符数组的Key类型，我们只要从模板参数列表中移走key参数，并将（放在尖括号中的）特化的类型添加到类名的后面，就可以创建一个特化的模板。

```
template <class Data>                   // remove the Key type
class DictEntry <char*>{                // append specialized type
   char*  key;                          // replace the Key type
   Data info;
public:
...} ;                                  // the rest of the class
```

讨论还没有结束。如果需要，还可以特化任意多个类型参数。当所有的参数被特化时，模板参数列表中的尖括号变空。例如下面的DictEntry类：

```
template < >                            // remove both parameter types
class DictEntry <char*, char*>{         // append specialized types
   char*  key;                          // replace the Key type
   char* info;                          // replace the Data type
public:
...} ;                                  // the rest of the class
```

如果只有第二个参数应该进行特殊处理，也没有问题，但必须在实际类型参数列表中重复第一个类型参数。

```
template <class Key>
class DictEntry <Key, char*> {
   Key  key;
   char* info;
public:
...} ;                                  // the rest of the class
```

当编译程序处理一个模板实例时，它选择适合其清单中最特殊者。如果没有找到实际类型的特化模板，则使用一般的模板类生成对象。

当某个成员类型需要特殊处理时，使用特化经常是很有必要的。使用特化将导致程序变得更庞大、更为复杂和更难于理解。并不是所有的编译程序都能很好地支持特化类。当某个数据类型需要特殊处理时（大多数情况下是字符数组），考虑用其他的名字编写一个单独的类，例如CharArray。编写一个单独的类的好处是用来初始化对象的类是确定的。其不足是不能保证在不同的特殊类中以相似的方法处理相似的特性。

有时候，很难做权衡。C++特化类提供了解决问题的一种途径。程序员自己决定如何使

用它。

17.6 模板函数

一个单独的非成员函数可以定义为一个模板函数，其语法与模板类成员函数的语法大致相同。

```
template <class T>
void swap(T& x, T& y)
{ T a = x;  x = y;  y = a; }
```

当函数需要原型时，它也包含模板参数列表，其中每一个class关键字后面都跟着一个参数。

```
template <class T> void swap(T& x, T& y);
```

在函数定义和函数原型（事先声明）中都要以template关键字开头，其后是用尖括号括起来的形式参数列表。每个形式参数由关键字class及其后面的程序员定义的标识符组成。关键字class和标识符用逗号隔开。标识符必须且只能在参数表中出现一次。

```
template <class T, class T>                    // this is illegal
void swap(T& x, T& y)
{ T a = x;  x = y;  y = a; }
```

每个类型参数都必须在模板函数的参数列表中使用。如果类型参数不在参数列表中，编译程序将认为是语法错误。

```
template <class T>
  int isEmpty(void);       // compile-time error for global function
```

与非模板函数相似，模板函数可声明为extern、inline或static；修饰符（如果有）紧接在形式参数的模板列表之后，出现在函数返回类型之前。

```
template <class T>
inline void swap(T& x, T& y)                   // inline function
{ T a = x;  x = y;  y = a; }
```

当编译程序处理一个模板函数定义时，并不生成目标代码。模板函数在调用时实例化。由于在参数列表中用名字提及了每个实际参数，编译程序也知道其类型，因而没有必要在调用模板函数时指定实参类型。

```
int a=5, b=10;  double c=3.0, d=4.0;
swap(a,b);                          // instantiation for integers
swap(c,d);                          // instantiation for double
```

由于编译程序已知道第一个函数调用的实际参数a和b的类型及第二个调用的实际参数c和d的类型，它将为swap(int&,int&)和swap(double&,double&)生成代码。

对返回值没有考虑参数匹配，如果需要可以进行转换。但模板参数不使用隐式转换。如果编译程序无法决定使用哪个函数以产生真正匹配的参数，将出现语法错误。

```
swap(a,c);                          // syntax error: no exact match
```

我们还可以重载模板函数，但要求通过实际参数类型的不同或实际参数的个数不同来区别这些函数。

```
template <class T>
inline void swap(T& x, T& y, T& z)        // three parameters
{ T a = x;  x = y;  y = z;  z = a; }
```

该函数可以与带有两个参数的swap()函数区别开来。

```
int a=5, b=10, c=20;
swap(a,b);  swap(a,b,c);
```

模板函数可为某特定类型进行特化。例如，字符数组不能与整数交换，必须使用一个已特化了的版本。函数特化的形成规则与模板类特化的规则相同。删除模板参数列表，将尖括号中的实际类型在函数名和参数列表之间移动。下面是特化的swap()函数。

```
template < >
inline void swap <char*> (char* x, char* y)
{ char* a = new char[strlen(x)+1];
  char* b = new char[strlen(y)+1];
  if (b==NULL) { cout << "Out of memory\n";  exit(1); }
  strcpy(a,x);  strcpy(b,y);              // caller must assure space
  strcpy(x,b);  strcpy(y,a);
  delete a;  delete b; }
```

客户代码为：

```
char x[20]="Hello!", y[20]="Hi, there!";  int a=5, b=10;
swap(a,b);          // general template function is instantiated
swap(x,y);          // specialized template function is instantiated
```

编译程序首先查找一个非模板函数，如果找到了一个且参数完全匹配，则不考虑模板。如果找到了多个非模板函数，则会出现语法错误。

如果没有找到匹配的非模板函数，则对模板函数进行考查。如果已存在一个完全匹配的模板实例，则使用该实例而不生成新的目标代码；否则，函数被实例化。如果找到了多个匹配，则会出现语法错误。

如果没有匹配的模板函数，则使用隐式转换，放松对参数的匹配要求，考查非模板函数。

模板函数不能由非模板函数调用，也不能作为参数传递给非模板函数。

17.7 小结

本章我们讨论了一种强有力的代码重用工具：C++模板。其基本思想很简单也很有吸引力：如果不同类型的对象使用相同的算法，则只需要编写一次算法，在使用时再指定算法应用的实际类型即可。

这是理想情况，在实际使用时会面临困难。C++模板的语法比较复杂，特化的使用甚至使问题更为复杂化了。有时候，研究在某种情况下使用哪个特化模板也是繁琐的事情。有时候，在一个编译程序和一台计算机上可行的代码可能在不同的编译程序和计算机上不可行。

此外，使用模板会增加程序的长度，对程序的性能也有不良影响。因此许多C++程序员都避免使用模板。另一方面，标准模板库（STL）中使用了模板。为了正确处理STL库，有必要理解并掌握使用模板的基本规则。

模板是个强有力的工具，但要小心使用它。

第18章 带异常处理的程序设计

本章我们将讨论一个比较新的C++论题：带有异常的程序设计。异常语言机制允许程序员将描述主要处理情况的源代码和描述例外情况的源代码分开。异常情况在正常的处理中不应该发生，但是偶尔会发生。将异常处理与程序中的主要处理情况区别开来，让主流情况和异常情况都更加容易阅读和维护。

这个定义有些模糊，不是吗？它有多种解释的可能性。实际上有些人认为属于异常或不正常的情况，其他人则认为是正常的系统操作部分。例如，我们在堆中分配内存时，算法应该描述了满足要求后的处理情况。由于计算机可能出现内存不足，该算法也应描述内存不足时的处理情况。那么内存不足对于该算法而言是否为异常呢？大多数人都认为是。

类似地，当程序从在线用户交互读取数据时，算法描述了对有效数据的处理。如果用户犯了错误输入无效的数据怎么办呢？这是一个异常吗？大多数人认为不是。他们认为在线错误是生活中经常出现的情况，处理这些错误的算法应是基本系统功能的一部分，而不应当看做很少出现的异常。

类似地，在循环中从一个文件读取数据时，算法指定了在读入下一个记录时要做的事情，即如何处理记录的各个部分。由于文件中有可能没有更多记录，算法应定义没有记录读取时的处理情况。那么到了文件末尾是否为异常呢？大多数人认为不是，到文件末尾只是标志着一个处理阶段（读取文件数据）的结束及下一个处理阶段（在内存中计算数据）的开始。

无论程序员是将实际问题看做带有一些额外异常情况的主流处理（第一个例子），还是看做重要性类似的不同情况的集合（第二、第三个例子），将不同计算任务的源代码混合在一起的问题是确实存在而且是大量的。

为了更好地构造算法，我们应该对程序设计语言中的一些工具进行全面的了解。因此我们首先介绍（作为C++程序设计技术的）异常是什么、程序员使用它的语法是什么，如何正确地使用它们，以及应该避免哪些不正确的使用。

最初，C++是不支持异常的，而是依赖C处理异常的机制，使用整个程序都可以访问的全局变量（例如errno）或者跳转去调用一个特殊的函数，该函数的名字是固定的，但是其内容可以由程序员定义（例如setjmp和longjmp）。

C++的异常处理是一种相当新的语言机制。与C++模板类似，异常处理机制比较复杂，使用异常的经验十分有限，而且它对系统设计的优点也还没有得到足够的验证。而且使用异常会增加程序的执行时间及可执行程序的规模。因此，我们建议不要在任何时候都使用异常。但是异常最终应该成为我们程序设计工具集中的一个合法部分。

18.1 异常处理的一个简单例子

通常，处理算法使用C++的流控制语句将正常的数据处理与错误处理或错误数据分隔开来，其中用得最多的是if或者switch语句。对多步骤算法而言，将主算法的源代码段和异常情况的代码段写在同一源程序的不同分支里，这通常使程序难于阅读，主要的代码行淹没

在众多异常处理和很少执行的代码中。

当在一个函数中出错时，它可能不知道如何处理错误。大多数情况下，终止程序运行可能在许多情形中是个好的解决方法。例如，将某个数据项压到一个系统栈时结果栈已满，这时就终止程序。另一方面，终止程序可能不能释放程序所占有的资源，如打开的文件或数据库锁等。

另一种办法是设置一个错误编码或返回一个错误值供调用者处理错误。例如，当客户代码想从一个空栈中弹出一个数据项时，返回一个错误值可能是个吸引人的解决方法。但这并不总是可行的。如果某给定类型的任意值对于弹出函数都是合法的，则找不出某个特别的值作为返回值提示调用者有异常情况。

当这种方法可行时，客户代码必须负责经常检查可能的错误。这样将使整个程序变得庞大，产生糟糕的客户代码，并降低了程序的执行速度。一般来说，这种方法容易出错。对于有些函数如C++的构造函数，没有返回值，则不能使用这种方法。

设置一个如errno的全局变量以表示错误的方法不适用于并发程序；对于串行程序而言，也难以一致地实现这种方法，因为它要求客户代码不停地检查全局变量。这些检查语句都堆砌在客户代码中，使程序更难于理解。

使用诸如setjmp和longjmp的库函数，程序可将控制转移给某个事件，让该事件负责释放外部资源并处理错误恢复。但在释放栈资源时可能没有为栈中创建的对象调用析构函数。因此，这些对象所占有的资源不能正常释放。

下面我们考虑一个简单例子，回顾一下异常处理技术应该解决的问题。程序18-1交互地提示用户输入分数的分子和分母，并计算打印分数值。为了计算结果，程序使用了两个服务器函数：inverse()和fraction()。第一个函数返回其参数的倒数，它由第二个函数fraction()调用。fraction()函数将其第一个参数与inverse()函数的返回值相乘。

程序18-1 在客户代码中处理错误的程序例子

```
#include <iostream>
using namespace std;

inline void inverse(long value, double& answer)
{ answer = 1.0/value; }                                  // answer = 1/value

inline void fraction (long numer,long denom,double& result)
{ inverse(denom, result);                                // result = 1.0 / denom
  result = numer * result; }                             // result = numer/denom

int main()
{
  while (true)                                           // infinite loop
  { long numer, denom; double ans;                       // numerator and denominator
    cout << "Enter numerator and positive\n"
         << "denominator (any letter to quit): ";
    if ((cin >> numer >> denom) == 0) break;             // enter data
    if (denom > 0) {                                     // correct input
      fraction(numer,denom,ans);                         // compute result
      cout << "Value of the fraction:          " << ans <<"\n\n";
      }
    else if (denom == 0)                                 // invalid result
      cout << "\nZero denominator is not allowed\n\n";
```

```
    else
      cout << "\nNegative denominator: " << denom <<"\n\n"; }
  return 0;
  }
```

当然，对于这个简单的计算问题而言，该设计显得复杂了。但是一个更加简单的设计不能演示异常处理的不同选择。一个更加复杂的问题可能适合用更加复杂的设计，但是会让我们过于关注细节。

在这个问题中，分母不能为0，否则拒绝并返回一条消息。也不允许分母为负数：如果分数是一个负数，则应是分子为负。如果分母为负数，则拒绝并打印出这个负数。

直到用户输入一个字符而不是数值时输入循环才停止，cout语句返回0，break语句终止循环。程序的输出结果如图18-1所示。

```
Enter numerator and positive
denominator (any letter to quit): 21 42
Value of the fraction:      0.5

Enter numerator and positive
denominator (any letter to quit): 21 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 42 -70

Negative denominator: -70

Enter numerator and positive
denominator (any letter to quit): 42 70
Value of the fraction:      0.6

Enter numerator and positive
denominator (any letter to quit): exit
```

图18-1　程序18-1的输出结果

在这个例子中，两个异常情况（分母为0和分母为负数）都是在客户代码中发现的，且错误在发现的地方立即被处理。服务器函数inverse()和fraction()没有机会处理错误的输入数据，因此无条件地计算它们的输出，而不对输入数据的有效性进行测试。

这里的错误恢复是打印一条消息，并重复要求输入下一个数据。这里的主流处理代码（调用fraction()服务器函数）没有与错误处理代码分开，但其后果并不严重。

通常，只有在进行某些处理后才在服务器代码中发现有错误，发现错误的地方与真正产生错误的地方相隔甚远。其中一些错误可在发现它们的地方进行处理，但有些错误处理可能需要一些其他信息，而发现错误的服务器函数又不知道这些信息。这时，有关错误的信息应当传回给该客户代码处理，若有可能还要进行恢复。为了模仿这种情形，我们将对输入数据的测试从客户代码转移到服务器函数inverse()中。

程序18-2演示了这种对错误的处理方法。inverse()函数计算其参数的倒数。如果参数为0，inverse()函数将用DBL_MAX常量（在头文件cfloat或float.h中定义的）作为倒数值，然后检查结果的有效性，并告诉其调用者在调用过程中发生的事情。

程序18-2 由服务器代码发现错误的程序例子

```
#include <iostream>
#include <cfloat>
using namespace std;

inline long inverse(long value, double& answer)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
    { cout << "\nZero denominator is not allowed\n\n";
        return 0; }                                          // zero denominator
  else if (value < 0)
      { return value; }                                      // negative denominator
  else
    return 1; }                                              // valid denominator

inline long fraction (long n,long d,double& result,char* &msg)
{ long ret = inverse(d, result);                             // result = 1.0 / d
  if (ret == 1)                                              // valid denominator
    { result = n * result; }                                 // result = n / d
  if (ret < 0)
        msg = "\nNegative denominator: ";
  return ret; }
int main()
{
  while (true)
  { long numer, denom;  double ans;                          // numerator/denominator
    char *msg;  long ret;                                    // error information
    cout << "Enter numerator and positive\n"
         << "denominator (any letter to quit): ";
    if ((cin >> numer >> denom) == 0) break;                 // enter data
      ret = fraction(numer,denom,ans,msg);                   // compute answer
    if (ret == 1)                                            // valid answer
      cout << "Value of the fraction:  " << ans <<"\n\n";
    else if (ret < 0)
      cout << msg << ret << "\n\n"; }                        // negative value
  return 0;
  }
```

如果结果是DBL_MAX，inverse()函数将打印一条错误消息，并返回0来告诉其调用者有关错误情况。如果参数为负数，inverse()函数处理这个错误，它返回这个负数让其客户识别并处理错误。否则，inverse()返回1，这将告诉调用者形式参数answer的值是有效的。

fraction()函数对inverse()的返回值进行判断：如果是1（有效结果），它将计算分数的值；如果是负数（负数分母），则将该负数传递给它自己的客户代码，并发送给该客户代码额外的用于错误处理的数据（要打印的消息）。客户代码对fraction()函数的返回值进行判断：如果是1，结果有效，主函数将显示这个结果；如果是负数，则打印这个负数及从fraction()函数接收的消息。否则，客户代码什么也不做，因为inverse()中已对错误（分母为0）做了处理。程序18-2的运行示例结果如图18-2所示。

可以看出，错误的发现与错误的处理在不同的地方，这会使解决方案更加复杂。服务器函数要处理额外的返回值和参数——强耦合度使不同的程序部分相互依赖，客户代码必须遵守有关返回值的复杂规定（在这个例子中，返回1表示参数值有效；返回0或负数表示参数值

无效），并根据不同的返回值进行不同的处理。这将使客户代码更加复杂，而且需要额外的文档让客户端代码程序员和服务器端代码程序员能成功地使用共同的规则。

```
Enter numerator and positive
denominator (any letter to quit): 42 0

Zero denominator is not allowed


Enter numerator and positive
denominator (any letter to quit): 42 -21

Negative denominator: -21

Enter numerator and positive
denominator (any letter to quit): -42 21
Value of the fraction:      -2

Enter numerator and positive
denominator (any letter to quit): exit
```

图18-2　程序18-2的输出结果

该设计的另一个问题是服务器代码inverse()函数和fraction()函数不仅要发现错误，还要就错误的原因与用户进行通信。对于这个只有三个函数的简单例子而言，这也许不是重大问题。但在比较复杂的程序中，确保每个函数只负责处理一个功能很重要。计算参数倒数的函数应知道如何计算参数的倒数，而不应参与用户界面。负责用户界面的函数应知道告诉用户什么内容，而不应涉及其他计算操作。这些任务应当分开。

该设计中还有一个问题是用户界面的成员分散在程序的所有代码中。当程序重新打包为德语、西班牙语、俄语或其他语言时，查询专线中没有特定的需要修改的地方，因为程序的每个地方都要进行修改。这是自找麻烦。

程序18-3试图弥补后面的两个不足。它可以作为使用静态数据成员和静态成员函数的例子。程序中用到的所有的输出字符串都放到MSG类中，作为一个私有的静态字符串数组。程序使用的所有输出字符串都作为一个私有的静态字符串数组移到类MSG中。该类提供了一个公共的静态函数msg()，该函数的参数表明了被使用的字符串的下标。如果下标不正确，则产生错误消息而不是所期望的信息。

程序18-3　客户代码和服务器代码之间大量通信的例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                          // internal static data
public:
      static char* msg(int n)                        // public static method
      { if (n<1 || n > 5)                            // check index validity
          return data[0];
        else
          return data[n]; }                          // return valid string
```

```
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",              // depository of text
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:          "
 } ;

inline long inverse(long value, double& answer, char* &msg)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
    { msg = MSG::msg(1);
      return 0; }                                     // zero denominator
  else if (value < 0)
       { msg = MSG::msg(2);
       return value; }                                // negative denominator
  else
       return 1; }                                    // valid denominator

inline long fraction (long n,long d,double& result,char* &msg)
{ long ret = inverse(d, result,msg);                  // result = 1.0 / d
  if (ret == 1)                                       // valid denominator
    { result = n * result; }                          // result = n / d
  return ret; }
int main()
{
  while (true)
  { long numer, denom;  double ans;                   // numerator/denominator
    char *msg;  long ret;                             // error information
    cout << MSG::msg(3) << MSG::msg(4);               // prompt user for data
    if ((cin >> numer >> denom) == 0) break;          // enter data
       ret = fraction(numer,denom,ans,msg);           // compute answer
    if (ret == 1)
      cout << MSG::msg(5) << ans <<"\n\n";            // valid answer
    else if (ret == 0)
      cout << msg;                                    // zero denominator
    else
      cout << msg << ret << "\n\n"; }                 // negative value
  return 0;
  }
```

服务器函数不再处理用户界面。虽然分析情况的代码仍然存在，但是这没法避免。如果代码需要发现错误，它应该检查一些相关的值，这就会让代码更加模糊。

用户界面的所有成员被放到一个地方。这不仅有助于将程序修改为其他语言，而且有助于用户界面的统一维护。如果要修改某个用户提示，则只修改MSG类。如果要增加或删除一个消息，则对静态数组MSG::data[]进行编辑，MSG::msg()方法中的数组成员个数也要相应地修改。为了避免修改MSG::msg()方法中的数组成员个数，数组成员个数（在msg()中定义为局部变量）可由sizeof(data)/sizeof(char*)来计算。由于消息的个数只使用一次，不妨将它设为一个字符值。

注意使用静态数据和静态方法的有关事项：static关键字、在类作用域之外初始化数据、在初始化语句和静态函数调用中使用类名、应用程序中缺少MSG类对象、函数MSG::msg()

和客户代码中的局部变量msg之间没有名字冲突。

该版本的程序输出结果与前面两个版本的应用程序的输出结果相同。因此，这里不再显示。

可以看出，将inverse()函数的任务限制为发现错误，而将错误处理（这里是打印有数据的信息）的任务转移，这样增加了客户和服务器之间的耦合度。在程序18-3中，inverse()函数多了一个参数，该参数由其客户fraction()函数传递给fraction()函数的客户main()。当分母为0时，只报告分母为0这个事实，这个事实由inverse()函数的参数msg传递上去。当分母为负数时，还要报告分母的值，inverse()函数使用其参数msg和这个返回值与它的调用者（及调用者的调用者）进行通信。

这里使用了额外的参数、返回值、复杂的调用规则等，C++的异常处理机制正是有助于免除这些麻烦。

18.2 C++异常的语法

C++异常允许程序员在某些事件（如错误）发生时改变控制流。这些错误（如没有找到文件、无效下标等）常常发生在运行时。当C++引发（raise）异常时，如果不知道如何处理异常，程序将终止运行。

异常处理机制（exception handler）是发生异常时执行的程序代码段。例如，给用户打印消息、收集信息用来分析异常的原因或进行错误恢复等。

与错误相关的源代码组织到异常处理机制中，可以使控制流更合乎逻辑。不再将所有的错误检查包含在主流代码中，因为那样做会导致算法意义难以理解，而是将错误处理编码到单独的地方。这种方法的缺点是，可能使发现错误的服务器函数意义晦涩。

我们可以将错误恢复代码分开来放在引起异常的同一函数中，也可以放到该函数的调用者中，还可以放在该调用者的调用者中等。这种灵活性使得带有异常的设计更为复杂。但异常处理机制允许程序员以规范的方式转移恢复操作的控制权。

总的来说，C++异常允许程序员将异常情况隔离开来，以远离源代码，从而使基本处理连成一个整体。这应该会让程序可读性更强，因而更加容易理解。但是情况不一定如此。正如前面提到的，使用异常的实用程序只是消除额外的参数、返回值，以及发现问题的函数与试图从问题中恢复过来的函数之间的复杂调用规则。

当C++引发或“抛出”（throw）一个异常时，可创建一个预先已定义的Exception类的对象或程序员定义类的对象。程序员定义的类可由Exception类派生而来，也可以是个独立的类。这种灵活性又让带异常的设计更加开放和难以理解。

异常可使用throw语句显式抛出，也可以是某个非法或无效行为的结果。异常用catch语句来捕获，控制转移到捕获异常的语句。捕获语句（或语句块）实施错误恢复（如果有的话）。在捕获语句块中错误恢复后的返回控制依赖于程序的结构。通常，控制不会自己返回到异常发生的地方。因此，如果需要有这样的返回（例如继续处理），程序员要以某种方式对程序进行构造。

下面是与异常处理相关的三个操作：

- 抛出异常。
- 捕获异常。

• 声明异常。

抛出异常指的是指出发现了某些异常（可能是错误的）情况，而且要求这些异常情况应使用C++的异常处理机制而不是一般的控制流技术来处理。

捕获异常指的是指定一段代码来响应某个特定的异常情况，而不响应其他的异常情况。

声明异常指的是在这个方法中指定可能引发的异常，它有助于编译程序（及客户端代码程序员和维护人员）了解函数的功能及其使用方法。

18.2.1 抛出异常

在抛出一个异常时，要使用关键字throw。它的作用是表明服务器代码已发现了一个不知如何处理的情况，于是抛出异常，希望别的地方（其客户或其客户的客户）有一段代码知道如何处理这个异常情况。

关键字throw用在抛出语句中。它的一般语法形式包括关键字throw及一个操作数，这个操作数可为任意类型的数据，用来寻找该异常处理机制。

```
throw value;
```

throw语句通常在程序中对某些数据或关系进行测试时，发现它们不能满足要求才执行。也就是说，服务器代码执行throw语句，将服务器代码中发现的问题通知其客户。

throw语句只能带一个任意类型的操作数。但有些编译程序并不标注带有多个操作数的throw语句为语法错误。（试图处理异常的）客户代码使用throw操作数获取错误上下文的相关信息。通常，这些信息用来定义客户代码在错误恢复时的行为。

下面是一个修改了的inverse()函数。在程序18-3中，该函数用返回值或参数值与客户代码进行通信。在这个版本中，inverse()函数在两种情况下抛出异常：1）如果发现分母为0，2）如果发现分母为负数。

```
inline void inverse(long value,double& answer)      // two parameters
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
        throw MSG::msg(1);                          // zero denominator
  if (value < 0)
        throw value; }                              // negative denominator
```

可以看出，对于0分母，函数抛出了一个字符数组类型的数据；对于负数分母，函数抛出了一个long类型的数据。这两个数据类型不同并非偶然。如果两个throw语句抛出的是同一类型的数据，异常处理更为困难。如果这两个异常都以相同的方式处理，不会有问题。但如果这两个异常必须按不同的方式处理，客户代码必须弄清楚抛出异常的服务器代码中究竟发生了什么事。

如果将这个inverse()函数与程序18-1中的版本进行比较，可以看出它们的界面是类似的：两个函数都返回了一个void类型的值，并只有两个参数。在程序18-1中，inverse()函数没有去发现错误，其客户的函数fraction()也没有去发现错误，而是main()客户代码发现了两个异常（0分母和负数分母）并进行了处理。

在程序18-2和程序18-3中，inverse()函数和fraction()函数都试图去发现异常，并从一些异常（0分母）恢复过来，而将剩下的异常（负数分母）留给main()客户处理。这样做使得代码耦合度很高并且比较复杂。上一个版本的inverse()函数抛出了两个异常，

还有一些分析代码（决定抛出什么异常），但其界面与程序18-1中的第一个版本一样简单。简单的界面使我们在捕获和声明这些异常时要多写一些代码。

18.2.2 捕获异常

抛出两个异常的inverse()函数有一个直接的客户和一个间接客户。直接的客户是调用inverse()函数的fraction()函数，间接的客户是调用fraction()函数的main()函数。一般而言，调用的层次没有限制。如果一个函数（例如本例中的inverse()）抛出了异常，但它本身并不处理这个异常，其调用者之一（直接的或间接的）必须捕获这个异常。

捕获异常是寻找能处理该错误的程序代码（异常处理机制）的过程，是通过对函数调用链的搜索来完成的。

人们可能认为捕获异常时必须使用关键字catch，这是正确的，C++的确有关键字catch，且用此关键字来捕获异常，但这还不够。当一个函数捕获异常时，它不能从任意的异常源中去捕获，而必须指明将试图从哪个程序代码段中去捕获异常。这就要求使用另一个C++关键字try，这个关键字后必须是能抛出异常的程序块。

负责捕获错误的客户代码包含了在try语句中引发异常的代码。

```
void foo()                    // function that catches exceptions
{ try                         // the try statement
    { statements; }           // statements that throw exceptions
  ...}                        // the rest of foo() with catch blocks
```

使用关键字try和catch实现C++的异常处理机制，抛出异常的语句（或方法调用）放在try语句块中，异常处理本身则放到catch语句块中。

从语法上而言，一个或多个catch语句块应该跟在一个try语句块后。每个catch语句块有一个类型参数对应于它处理的异常。

```
void foo()                        // function that catches exceptions
{ try
    { statements; }               // statements that throw exceptions
   catch (Type1 t1)               // catch block for thrown type Type1
    { handler_for_Type1(); }
   catch (Type2 t2)               // catch block for thrown type Type2
    { handler_for_Type2(); }
        . . . . .
   catch (TypeN tN)               // catch block for thrown type TypeN
    { handler_for_TypeN(); }
  statements_executed_after_the_try_or_catch_block; }
```

try语句后面至少有一个catch语句块提供异常处理。一个catch语句块前没有try语句是错误的。（如果此catch语句块和try语句之间还有其他catch语句则是可以的）一个try语句后没有catch语句块也是错误的。

前面曾经提到过，throw语句有个参数，该参数可以是字符数组、长整型数据，甚至可以是程序员定义类的类型值。该参数值通常含有表示错误上下文的信息。在inverse()函数中，这些数据或者是要打印的消息字符串，或者是要显示的负数分母。如果throw语句抛出了下一个类型的对象，该类型的构造函数应使这个对象带有与问题相关的信息。这些信息可能由catch语句用来诊断或进行错误恢复。

如果在try语句块后有几个catch结构，这些catch语句块必须有不同类型的参数。由于catch语句块没有名字来标识，因此它们的参数类型必须互不相同。

如果try语句块中抛出的异常类型与某个catch语句块中的参数相匹配，则执行该catch语句块中的代码，并停止搜索。当该catch语句块终止后，将执行这个try语句中catch语句块之后的语句。

“参数匹配”指的是由该try语句块抛出的异常对象可赋给该catch块的参数，则意味着是完全匹配，也可以是任意标准的类型转换，或者抛出的异常对象是catch块参数的任意子类。例如，一个double数据可由带有long类型参数的catch语句块捕获，一个SavingsAccount对象可被带有Account类参数的catch语句块捕获。

在catch语句块终止后，执行try语句块及其catch结构后的语句。如果需要，这些语句中可以有其他的try语句块（后跟catch语句）。如果try语句没有抛出任何异常，其后的catch语句块则视为空语句，即跳过该语句。

如果在try语句的中间抛出了异常，try语句的执行将终止，查找并执行相应的catch语句。try语句块中抛出异常语句后的语句将永远不会执行。通常这是很符合逻辑的，因为这些语句不能执行才抛出该异常。

如果try语句块中的代码抛出了一个异常，但没有相应类型的catch语句会怎样呢？那就太糟糕了，函数终止执行。这样，不仅该try语句块不能完全执行，而且catch语句块后面的语句也不能执行。这意味着要在函数的客户代码中查找合适的catch语句块。如果找到了就好，如果一直搜索到main()中也找不到处理这个异常的catch语句，程序将终止执行。

例如，下面版本的inverse()函数throw抛出了异常并试图捕获它们。

```
inline void inverse(long value,double& answer)      // two parameters
{ try  // start of try block
   { if (value == 0)                                // zero denominator
        throw MSG::msg(1);
     if (value < 0)                                 // negative denominator
        throw value;
     answer = 1.0 / value; }                        // end of the try block
  catch (char* str)
   { cout << str; }                                 // zero denominator
  catch (long val)
   { cout << MSG::msg(2) << val << "\n\n"; }}       // negative value
```

如果inverse()函数的第一个参数是个合法的数据，将完整地执行try语句块，而跳过catch语句块。由于catch语句块后没有语句，因此函数终止，好像根本没有异常处理机制一样。

如果第一个参数是0，抛出字符数组异常，则执行第一个catch语句块。注意catch语句块是一个“块”——它有自己的作用域，其参照是它自己的参数str而不是实际抛出的变量MSG::msg(1)。

类似地，如果第一个参数是负数，则抛出这个负数，执行第二个catch语句块。同样，打印的值的名字是val而不是value。无论抛出了什么异常，answer = 1.0/value将永远也不会执行。这是合理的，因为只有当这个value经过了所有测试判断后才执行这条语句。

如果在inverse()函数中try语句块中抛出了异常，但没有相应的catch语句块处理，则在fraction()函数中继续查找相应的catch语句。如果仍没有，再到main()中查找。

在该版本中的inverse()函数中，throw语句和catch语句块在同一个函数作用域内。从语法角度而言，这在C++中是合法的。但从软件工程的角度而言，这样是多余的——如果与异常有关的信息在同一个函数中，则没有必要使用异常处理机制。这种情况下，在inverse()函数中使用简单的if语句就可以有同样的效果。

该版本的异常处理的另外一个问题与程序的其他部分的执行有关。在inverse()函数终止后，该函数的调用者fraction()函数和main()函数不知道是否有异常发生。同时，如果引发了异常，计算答案的语句将不能执行，inverse()函数的调用者应了解这些情况。

下面，我们再考虑inverse()的另一个版本，它抛出异常，但不捕获异常。

```
inline void inverse(long value,double& answer)      // two parameters
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
        throw MSG::msg(1);                          // zero denominator
  if (value < 0)
        throw value; }                              // negative denominator
```

我们在客户函数fraction()中试图捕获这些异常。

```
inline void fraction (long numer, long denom, double& result)
{ try {
   inverse(denom, result);                          // result = 1.0 / denom
   result = numer * result; }                       // result = numer / denom
  catch (char* str)
   { cout << str; }                                 // zero denominator
  catch (long val)
   { cout << MSG::msg(2) << val << "\n\n"; }}       // negative value
```

这个版本的inverse()函数并不比前面的版本好。我们应在客户代码中处理异常，并在那些有关异常的信息可以用来改变程序行为的地方处理异常。本例不显示计算的结果。

程序18-4演示了在main()函数中处理异常。正如我们在前面提到的，该例子中有人为因素，因为main()函数本身可以发现无效输入。假如main()函数不能发现无效输入，那么这个异常处理方案就变得很有意义了：inverse()函数发现了错误，并给main()发送信息，因此main()可以不使用无效结果。

程序18-4　throw及catch异常的例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                         // internal static data
public:
      static char* msg(int n)                       // public static method
      { if (n<1 || n > 5)                           // check index validity
          return data[0];
        else
          return data[n]; }                         // return valid string
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",            // depository of text
"\nNegative denominator: ",
```

```
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:          "
 } ;

inline void inverse(long value, double& answer)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
        throw MSG::msg(1);
  if (value < 0)
        throw value; }

inline void fraction (long numer, long denom, double& result)
{ inverse(denom, result);                          // result = 1.0 / denom
  result = numer * result; }                       // result = numer/denom

int main()
{
  while (true)
  { long numer, denom;  double ans;                // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);            // prompt user for data
    if ((cin >> numer >> denom) == 0) break;       // enter data
    try {
      fraction(numer,denom,ans);                   // compute answer
      cout << MSG::msg(5) << ans <<"\n\n";         // valid answer
      }
    catch (char* str)                              // zero denominator
    { cout << str; }
    catch (long val)                               // negative value
    { cout << MSG::msg(2) << val << "\n\n"; }
  }
  return 0;
  }
```

在程序18-4中，inverse()函数分析情况，并为其调用者抛出了两个异常。该函数的直接调用者fraction()函数没有任何异常处理机制（catch语句块），因为忽略输出的语句放在main()函数中。由于fraction()函数没有任何catch语句块，所以也没有try语句，因为无catch语句块的try语句是不合法的。

如果inverse()函数没有抛出异常，fraction()函数和main()函数将继续计算、打印结果、要求输入下一个数据。如果inverse()函数抛出了一个异常，这个异常不能在inverse()函数中进行处理，因为该函数没有合适的catch语句。这样将对fraction()函数进行搜索，由于fraction()函数也没有任何异常处理机制，因而搜索main()函数。如果main()函数仍没有任何异常处理机制，程序将终止执行。

当搜索到main()函数时，发现了try语句和catch语句块。对于main()函数而言，其服务器函数fraction()是问题源。客户main()函数并不关心是fraction()函数是从它的服务器中接收一个异常还是fraction()函数自己抛出了一个异常。如果fraction()函数抛出了一个异常，try语句将在显示答案之前终止执行，相应的异常处理机制打印一条使用了inverse()产生的信息的消息。

try语句块应该有多大呢？在这个例子中，try由两条语句组成：调用fraction()函数的语句和输出语句。如果我们将调用fraction()函数的语句移到try语句之外会怎样呢？

```
int main()
{ while (true)
  { long numer, denom;  double ans;                    // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);                // prompt user for data
    if ((cin >> numer >> denom) == 0) break;           // enter data
    fraction(numer,denom,ans);                         // compute answer
    try {
      cout << MSG::msg(5) << ans <<"\n\n"; }           // valid answer
    catch (char* str)                                  // zero denominator
    { cout << str; }
    catch (long val)                                   // negative value
    { cout << MSG::msg(2) << val << "\n\n"; } }        // end of loop
    return 0; }
```

这样就无法达到目的，try语句将不会抛出任何异常，catch语句块也捕获不到异常，它们只能处理前面的try语句中发生的异常，当inverse()函数给fraction()函数抛出一个异常时，fraction()函数将给main()函数抛出这个异常，没有catch语句块将处理这个异常，程序将终止执行。

如果只将函数调用放在try语句中，而将输出语句移走又会怎样呢？这样做的理论是既然该语句不抛出任何异常，它就浪费了try语句的宝贵空间。

```
int main()
{ while (true)
  { long numer, denom;  double ans;                    // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);                // prompt user for data
    if ((cin >> numer >> denom) == 0) break;           // enter data
    try {
     fraction(numer,denom,ans); }                      // compute answer
    cout << MSG::msg(5) << ans <<"\n\n";               // valid answer
    catch (char* str)                                  // zero denominator
    { cout << str; }
    catch (long val)                                   // negative value
    { cout << MSG::msg(2) << val << "\n\n"; } }        // end of loop
    return 0; }
```

这段代码将产生语法错误。输出语句现在位于try语句和catch语句块之间，使得catch语句块不能直接跟在try语句的后面。更糟糕的是，catch语句块没有紧接在try语句后面，编译程序会提示这个不好的消息。

如果将try语句进行扩充，使它再包含一个循环语句会如何呢？这样做的理论可能是将不同的异常源放在一起，在同一catch语句块的缓冲区中对它们进行处理。

```
int main()
{ while (true)
  { long numer, denom;  double ans;                    // numerator/denominator
   try {
     cout << MSG::msg(3) << MSG::msg(4);               // prompt user for data
     if ((cin >> numer >> denom) == 0) break;          // enter data
     fraction(numer,denom,ans);                        // compute answer
     cout << MSG::msg(5) << ans <<"\n\n"; }            // end of try
    catch (char* str)                                  // zero denominator
    { cout << str; }
    catch (long val)                                   // negative value
    { cout << MSG::msg(2) << val << "\n\n"; } }        // end of loop
    return 0; }
```

这样是可行的，尤其当客户代码中产生了其他异常时将会很有用。但一般来说，try语句的作用域应尽可能的小，使维护人员容易辨别异常的来源。

如果将整个while循环放在try语句中又会如何呢？这将依赖于如何处理。如果只是将try关键字及开花括号上移，而将闭花括号保持原位不动，编译程序将不会接受。

```
int main()
{ try {
  while (true)
  {  long numer, denom;  double ans;                    // numerator/denominator
     cout << MSG::msg(3) << MSG::msg(4);                // prompt user for data
     if ((cin >> numer >> denom) == 0) break;           // enter data
     fraction(numer,denom,ans);                         // compute answer
     cout << MSG::msg(5) << ans <<"\n\n"; }             // end of try
   catch (char* str)                                    // zero denominator
   { cout << str; }
   catch (long val)                                     // negative value
   { cout << MSG::msg(2) << val << "\n\n"; } }          // end of loop
   return 0; }
```

现在，try语句的作用域没有嵌套在while循环作用域之中。注意无论做什么决定，作用域之间的嵌套都要正确，否则编译程序将不会接受。总的来说，try语句的作用域越小越好。

如这些例子所示，在设计异常处理机制时要考虑三个基本问题：

- 在哪里抛出异常。
- 在哪里捕获异常。
- 给异常处理程序发送什么信息。

在本章的开始部分，我们提到使用异常的常用基本原理，即通过将主流的处理和异常情况的处理区分开来以简化客户代码。在本例中，这个原理的重要性反而是其次的。重要的是该客户代码充斥了try语句和catch结构及其参数和括号。

这正是用异常进行设计的另一个方法，我们在发现错误的地方抛出异常，并且收集错误恢复所需要的信息；在决定如何进行错误恢复的地方放置catch子句。在这个简单的例子中，恢复措施只是跳过答案显示的步骤。但是它仍然需要从发现错误的地方把数据发送到错误恢复的地方。

18.2.3 声明异常

*声明异常*指的是定义在函数中要抛出的异常，但函数本身不对这些异常进行处理。也就是说，定义函数将传送给其调用者的异常。如果函数本身不能捕获异常，而希望其他函数去处理，该函数必须对这些异常进行声明。

在声明异常时要使用throw关键字，其语法包括三个方面：常规的函数声明、throw关键字及（括号括起来的）类型列表。函数将抛出的这些类型的数据用于搜索异常处理程序。

```
functionDeclaration throw (Type1, Type2, ... TypeN);
```

在函数调用中出现了非法情况时，函数代码可隐式地给其服务器函数抛出异常，也可使用throw关键字显式地抛出异常。

如果函数代码抛出某个异常，而该函数本身又捕获这个异常，则不需要将该异常包含在

抛出列表中。如果服务器函数抛出并捕获了一个异常，那么该异常也不应包含在客户函数的抛出列表中。抛出列表中只包含这个函数的客户必须处理的异常。

例如，程序18-4中的inverse()函数显式地抛出了（但没有捕获）两个异常：一个字符数组和一个long类型。该函数的定义应包含throw关键字及这两种类型的异常。

```
inline void inverse(long value, double& answer)
                    throw (char*, long)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
        throw MSG::msg(1);                    // explicit throw
  if (value < 0)
        throw value; }                        // explicit throw
```

类似地，程序18-4中的fraction()函数并没有显式地抛出任何异常，但其服务器函数inverse()抛出了（没有捕获）两个异常。这意味着fraction()函数隐式地抛出了这两个异常，因此应对这两个异常进行声明。

```
inline void fraction (long numer, long denom, double& result)
                     throw (char*, long)
{ inverse(denom, result);                     // implicit throw
  result = numer * result; }                  // result = numer/denom
```

如果函数没有抛出任何异常，声明时可以使用空的throw定义throw()。例如，

```
void foo() throw ();                    // expect no exceptions
```

如果某个函数没有定义异常说明，那么该函数有可能抛出任意类型的异常。

```
void foo();                // no throw: expect any exception
```

一个函数声明了一个异常，但该异常并不真正抛出，若C++认为这种情况是错误的，那么情况就比较好办。如果没有声明一个函数真正抛出的异常也被认为是错误的，那么也比较好办。但是，实际情况并非如此，我们可以撤销伪装的声明（即函数不抛出所声明的异常），或者不恰当地声明（即只声明抛出的部分异常），或者干脆忽略这个问题，就像程序18-4所充分演示的那样。

理解程序中其他人的异常设计可能是一件很麻烦的事情，可能需要所有人的帮忙。声明异常是为代码的设计进行文档化的强大技术。一定要很好地使用它。

当函数只部分地处理异常时，要在函数声明异常时反映出来。程序18-5显示了如何声明函数inverse()和fraction()，并表示了两者之间的不同任务划分。inverse()函数抛出（并声明）了程序18-4中相同的异常，fraction()函数本身对long类型的异常进行了处理，这样，在fraction()函数的界面中只声明了一个字符数组类型的异常。

程序18-5　claim、throw与catch异常例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
       static char* data [];                          // internal static data
public:
       static char* msg(int n)                        // public static method
```

```
        { if (n<1 || n > 5)                              // check index validity
            return data[0];
          else
            return data[n]; }                            // return valid string
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",                 // depository of text
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:          "
 } ;

inline void inverse(long value, double& answer)
          throw (char*, long)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
        throw MSG::msg(1);
  if (value < 0)
        throw value; }

inline void fraction (long numer, long derom, double& result)
          throw (char*)
{ try {
    inverse(denom, result); }                            // result = 1.0 / denom
  catch (long val)                                       // negative value is OK
      { cout << MSG::msg(2) << val << "\n\n"; }
  result = numer * result; }                             // result = numer / denom

int main()
{
  while (true)
  { long numer, denom;  double ans;                      // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);                  // prompt user for data
    if ((cin >> numer >> denom) == 0) break;             // enter data
    try {
      fraction(numer,denom,ans);                         // compute answer
      cout << MSG::msg(5) << ans <<"\n\n"; }             // valid answer
    catch (char* str)                                    // zero denominator
    { cout << str; }
   }
  return 0;
  }
```

因此，main()函数只需要处理一个异常，而不是程序18-4中的两个异常。该程序的输出结果如图18-3所示。

该例子展示了在函数接口中声明异常的好处。当客户端代码程序员想知道客户函数要处理的异常时，只需检查该客户函数调用的所有服务器函数的声明就可以了。

18.2.4 重新抛出异常

注意图18-3中的程序执行情况与图18-2中的不同。在图18-2中，拒绝负数分母，并要求用户输入新的数据。在图18-3中，拒绝负数分母，但计算后的结果还是打印出来了。

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -11

Negative denominator: -11

Value of the fraction:      -1

Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:      -0.25

Enter numerator and positive
denominator (any letter to quit): quit
```

图18-3　程序18-5的输出结果

这是因为fraction()函数本身从这个异常中恢复过来（通过打印一条消息及分母的值），main()函数认为其结果是有效的，没有阻止输出。

这是一个相当普遍的情形，函数只是从异常中部分恢复过来，但其他的某些行为必须在调用者中完成。C++允许函数重新抛出异常来处理这种情况。只要在catch语句块中使用throw语句即可。

例如，inverse()函数通过再次抛出异常来告诉main()它并没有完全恢复异常。

```
inline void fraction (long numer, long denom, double& result)
          throw (char*, long)              // extra exception claim
{ try {
    inverse(denom, result); }              // result = 1.0 / denom
  catch (long val)
       { cout << MSG::msg(2) << val << "\n\n";
         throw val; }                      // throw it again
  result = numer * result; }
```

注意，这样不会导致无限地循环。在catch语句块中抛出的异常不能进入其作用域中，要想达到这样的效果，异常应该出现在catch结构之前的try语句块中。形式上来说，异常被认为是在它的异常处理机制的入口处处理的。因此，这个throw语句将要求在更高层次中，即调用fraction()函数的客户代码中搜索另一个long类型的异常处理程序。

再次抛出同一类型（和值）的异常的另一种办法是只在catch语句块中用throw关键字，这样，由catch语句参数所定义的异常将再次被抛出。

```
inline void fraction (long numer, long denom, double& result)
          throw (char*, long)              // extra exception claim
{ try {
    inverse(denom, result); }              // result = 1.0 / denom
  catch (long val)
       { cout << MSG::msg(2) << val << "\n\n";
         throw; }                          // same as "throw val"
  result = numer * result; }
```

程序18-6演示了这种方法。其中，inverse()函数与程序18-5中的相同，fraction()

函数处理了long类型异常的部分工作，然后再次抛出这个异常。这样fraction()函数必须在界面中声明该异常，main()函数必须提供catch子句捕获这个异常。如果main()函数中没有捕获这个异常的catch子句，程序将异常终止。

程序18-6 在catch语句块中重新抛出异常的例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
       static char* data [];                          // internal static data
public:
       static char* msg(int n)                        // public static method
       { if (n<1 || n > 5)                            // check index validity
           return data[0];
         else
           return data[n]; }                          // return valid string
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",              // depository of text
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:          "
 } ;

inline void inverse(long value, double& answer)
           throw (char*, long)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
         throw MSG::msg(1);
  if (value < 0)
         throw value; }
inline void fraction (long numer, long denom, double& result)
           throw (char*, long)
{ try {
    inverse(denom, result); }                         // result = 1.0 / denom
  catch (long val)                                    // negative value is OK
       { cout << MSG::msg(2) << val << "\n\n";
      throw val; }
  result = numer * result; }                          // result = numer / denom

int main()
{ cout << endl << endl;
  while (true)
  { long numer, denom;  double ans;                   // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);               // prompt user for data
    if ((cin >> numer >> denom) == 0) break;          // enter data
    try {
         fraction(numer,denom,ans);                   // compute answer
         cout << MSG::msg(5) << ans <<"\n\n"; }       // valid answer
    catch (char* str)                                 // zero denominator
    { cout << str; }
    catch (long)                                      // just type
       { } // empty body
```

```
    }
  return 0;
  }
```

由于再次抛出这个异常的惟一目的是避免在main()中显示结果，因而main()中捕获这个异常的catch语句块中没有任何处理，catch语句体是空的。但它必须在那里。为了避免出现没有使用catch语句的参数的警告，我们在参数表中省略了参数，只剩下了参数的类型。这虽然不太好，但这是合法的C++技巧。

程序的输出结果如图18-4所示，我们可以看到成功地遏止了附加的输出。

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -11

Negative denominator: -11

Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:      -0.25

Enter numerator and positive
denominator (any letter to quit): quit
```

图18-4　程序18-6的输出结果

这是让几个函数协同处理同一异常的有力方法。但要谨慎使用，因为该方法的出发点是将本该属于一个整体的异常处理拆分开来。当很难在一个地方集中处理异常时，程序员可尝试使用这种方法，会使程序编写更为简单，但极有可能让代码更难理解。

18.3　类对象的异常

在上面的诸例子中，throw语句不仅将控制权发送给catch语句块，另外还传递了某个特定类型的数据，在catch语句块中可以对这个值进行访问。这种重要的方法为发现错误的程序部分和错误恢复的程序部分建立了通信。

发送某个特定类型的数据既是一种特权（建立通信），也是一种限制。因为一个函数不能将同一类型的数据发送给不同的catch语句块处理。例如，一个函数在两个不同的地方抛出了两个不同的字符串，这两个字符串必须由同一catch语句块来处理。如果将错误恢复只限于打印消息，这将没有关系——同一catch语句块可以打印两条不同的消息。

```
void foo() throw (char*)
{ if (test1())
    throw "One bad thing happened";              // one problem
  else if (test2())
    throw "Another bad thing happened";          // another problem
  proceed_safely(); }                            // no problem

void client()
{ try
```

```
  { foo(); }                                    // no problem
catch(char* msg)
{ cout << msg << endl; } }                      // either problem
```

但是，如果程序对于不同的错误源有不同的处理时，这种传送数据的机制将变得非常刻板——catch语句块必须对throw语句传送的数据进行分析，并根据分析结果设置不同的处理分支。这与在不同的catch语句块中处理不同的错误的目标相违背。

这种异常处理机制的另一个固有局限是从try语句中只能发送一个数据值到catch语句块。当需要发送多个数据时，程序员不得不求助于其他策略了。从程序18-1到程序18-6所显示的例子中，对于负数分母的异常处理要求有两条信息：分母是负数这个事实及负数分母的值。我们将其中的一条信息（分母的值）作为参数传送给catch语句块，并使用一个全局字符数组来表示错误消息。

在C++中，通过抛出复合对象而不是简单的内部数据类型的异常值来解决上述问题。

18.3.1 抛出、声明与捕获对象的语法

抛出对象对C++程序设计添加了新的要求。设计者必须决定从发现错误的地方发送什么数据项到错误恢复的地方。对于每个异常，我们都要创建一个类，以携带有关错误的必要数据。这个类的诸方法应允许catch语句块具有访问其对象数据的能力。

例如，把ZeroDenom类设计为携带一个有关0分母的数据。在发现错误的地方可以创建并抛出这个类的对象。这个对象只需要一条信息（即消息），这条信息对于所有的错误都是相同的。因此，ZeroDenom类应提供一个缺省的构造函数。由于在catch语句块中要打印消息，ZeroDenom类还应提供一个可被catch语句块调用的print()方法。

```
class ZeroDenom {
  char *msg;                    // data to be carried to error handler
public:
  ZeroDenom ()                  // it is called by the throw statement
  { msg = MSG::msg(1); }
  void print () const           // it is called by the catch block
  { cout << msg; }
} ;
```

将类的对象作为异常信息的携带者时，与内部数据类型值的异常类似，要经过同样的三个步骤：1) 抛出异常，2) 捕获异常，3) 声明异常。

例如，发现异常情况的函数inverse()，创建这个类的对象，并在ctach块的查找中抛出该对象。

```
if (answer==DBL_MAX)
  throw ZeroDenom();                // unusual syntax
```

注意缺省构造函数调用的语法，是使用类名和一对空括号()。在其他情况下（如用new运算符创建一个对象），使用空括号为语法错误；但在这里，省略空括号则是语法错误。如果不习惯这种语法，可以创建一个所需类型的对象，然后再抛出该对象，就像抛出内部数据类型的变量一样。

```
if (answer==DBL_MAX)
  { ZeroDenom zd; throw zd; }       // conventional syntax
```

当使用非缺省的构造函数时，抛出该对象的语法与其他上下文的情况相同。例如，为了传递有关负数分母的信息，我们设计了NegativeDenom类，这个类有表示错误消息的数据成员、表示分母值的数据成员及访问该对象数据成员的方法。

```
class NegativeDenom {
        long val;                    // private data for exception info
        char* msg;
public:
  NegativeDenom(long value)          // conversion constructor
        : val(value), msg(MSG::msg(2)) { }
  char* getMsg() const
   { return msg; }
  long getVal() const                // public methods to access data
   { return val; }
} ;
```

为了抛出这种类型的对象，抛出对象的方法必须要指定构造函数的参数值，例如inverse()函数。

```
if (value < 0)                       // analyze the situation
  throw NegativeDenom(value);        // throw an exception
```

与没有参数的对象类似，也可以使用常规的语法创建一个对象，然后像抛出内部数据类型的变量一样抛出这个对象。

```
if (value < 0)
 {NegativeDenom nd(value); throw nd; }
```

声明对象异常的语法与内部数据类型的异常声明一样，但必须使用类名代替内部数据类型名。下面是inverse()函数声明了ZeroDenom类和NegativeDenom类的异常。

```
inline void inverse(long value, double& answer)
           throw (ZeroDenom, NegativeDenom)          // claim exceptions
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
    throw ZeroDenom();                               // throw class object
  if (value < 0)
    throw NegativeDenom(value); }                    // throw class object
```

为了捕获一个类的对象，catch语句块应定义这个类的参数。在catch语句块作用域中，访问这个对象的规则与访问任意其他类的对象一样。下面的客户main()函数捕获了两个异常。

```
try {
  fraction(numer,denom,ans);                          // compute answer
  cout << MSG::msg(5) << ans <<"\n\n"; }              // valid answer
catch (const ZeroDenom& zd)                           // zero denominator
 { zd.print(); }
catch (const NegativeDenom &nd)                       // negative value
 { cout << nd.getMsg() << nd.getVal() << "\n\n"; }
```

第一个catch语句块给对象发送一个消息，并让该对象打印消息；第二个catch语句块获取数据成员的值，并打印这些值。第一个方法当然好一些。第二个方法需要将NegativeDenom类的数据成员设为公共的。

程序18-7与程序18-1~程序18-6相同。inverse()函数抛出了ZeroDenom类和

NegativeDenom类的对象。由于该函数被fraction()函数调用，而fraction()函数并不知如何（从fraction()调用者的角度）处理这些异常，于是抛出了这些异常。fraction()函数也声明了这两个异常。因此main()在try语句块中调用fraction()函数，并提供了两个catch语句块，一个catch语句块对应一个异常。

程序18-7 抛出类对象的例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
       static char* data [];                    // internal static data
public:
       static char* msg(int n)                  // public static method
       { if (n<1 || n > 5)                      // check index validity
           return data[0];
         else
          return data[n]; }                     // return valid string
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",       // depository of text
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:        "
 } ;

class ZeroDenom {
       char *msg;                               // data to be carried to error handler
public:
       ZeroDenom ()                             // it is called by the throw statement
       { msg = MSG::msg(1); }
       void print () const                      // it is called by the catch block
       { cout << msg; }
} ;

class NegativeDenom {
    long val;                                   // private data for exception info
    char* msg;
public:
    NegativeDenom(long value)                   // conversion constructor
          : val(value), msg(MSG::msg(2)) { }
    char* getMsg() const
        { return msg; }
    long getVal() const                         // public methods to access data
        { return val; }
} ;

inline void inverse(long value, double& answer)
         throw (ZeroDenom, NegativeDenom)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
     throw ZeroDenom();
  if (value < 0)
```

```
    throw NegativeDenom(value); }

inline void fraction (long numer, long denom, double& result)
          throw (ZeroDenom, NegativeDenom)
{ inverse(denom, result);                              // result = 1.0 / denom
  result = numer * result; }                           // result = numer/denom

int main()
{
  while (true)
  { long numer, denom;  double ans;                    // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);                // prompt user for data
    if ((cin >> numer >> denom) == 0) break;           // enter data
    try {
      fraction(numer,denom,ans);                       // compute answer
      cout << MSG::msg(5) << ans <<"\n\n"; }           // valid answer
    catch (const ZeroDenom& zd)                        // zero denominator
    { zd.print(); }
    catch (const NegativeDenom &nd)                    // negative value
    { cout << nd.getMsg() << nd.getVal() << "\n\n"; }
   }
  return 0;
  }
```

18.3.2 使用带异常处理的继承

一个程序中引发出错的条件可能与另一个程序中的条件相类似，进行错误恢复所需的信息结构也可能大致相同。例如，程序18-7中每个异常都有一个指向字符数组的指针，字符数组中存放着要打印的错误消息。

与相似类一样，设计者可将程序中的异常类组织成一个继承层次结构。例如，可以重新设计ZeroDenom类和NegativeDenom类，使NegativeDenom类从ZeroDenom类派生而来。

```
class ZeroDenom {
protected:
       char *msg;
public:
       ZeroDenom (char* message) : msg(message)
       { }
       void print () const
       { cout << msg; }
} ;
```

在该类的前一个版本中，我们将字符数组的名字硬编码在类的构造函数中。因此，该类的客户即程序18-7中的函数inverse()不需要知道随异常传送的消息，它只需要使用缺省的构造函数创建异常对象就可以了。在这个版本中，ZeroDenom类不知道它的对象携带的信息，它的客户必须显式地指定携带的消息。

我们并不确定哪个方法更好一些。总的来说，（程序18-7实现的）第一个方法将任务推向服务器类ZeroDenom，而第二个方法将任务上托给客户类。但是在程序的类中分配任务的整体结构则第二个方法更加吸引人一些。不论在每个特定的情况下哪个方法更加好，我们都要确保没有忽视两者的不同，并且对程序中“哪部分知道什么”的问题保持敏感。

```
class NegativeDenom : public ZeroDenom {
        long val;
public:
        NegativeDenom(char *message, long value)
           : ZeroDenom(message), val(value) { }
        void print () const
        { cout << msg << val << "\n\n"; }
} ;
```

我们从ZeroDenom类派生NegativeDenom类，而不是从相反的方向。那么能否从NegativeDenom类派生ZeroDenom类呢？从理论上而言是可以的，但从实践观点看，这不是好主意。因为NegativeDenom类比ZeroDenom类包含更多的数据成员。

我们将基类ZeroDenom的数据成员定义为受保护的而不是私有的，于是派生类NegativeDenom能访问基类数据。如果ZeroDenom类的数据成员定义为私有的，NegativeDenom类中的方法必须使用ZeroDenom类的方法才能访问ZeroDenom类的数据成员。例如，NegativeDenom类可设计为如下形式：

```
class NegativeDenom : public ZeroDenom {
        long val;
public:
        NegativeDenom(char *message, long value)
           : ZeroDenom(message), val(value) { }
        void print () const
        { ZeroDenom::print();                    // call to the base method
           cout << val << "\n\n"; }
} ;
```

如果基类和派生类中的两个算法有共同的元素，在派生类代码中强调这一点比较好（即在相应的派生类方法中调用基类的方法）。另一方面，如果添加的基类访问方法只被派生类使用，那么这种方法就显得费时费力。

当异常类有继承关系时，声明异常对象和抛出异常对象的方法与无关的异常类相同。但是，捕获异常时如果不注意类之间的关系，可能会出现其他问题。我们对程序18-7修改一下，使NegativeDenom类从ZeroDenom类派生后如程序18-8所示。

程序18-8 使用继承关系的异常类例子

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
        static char* data [];                          // internal static data
public:
        static char* msg(int n)                        // public static method
        { if (n<1 || n > 5)                            // check index validity
             return data[0];
          else
             return data[n]; }                         // return valid string
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",               // depository of text
```

```
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction:          "
 } ;
class ZeroDenom {
protected:
       char *msg;
public:
       ZeroDenom (char* message) : msg(message)
       { }
       void print () const
       { cout << msg; }
} ;

class NegativeDenom : public ZeroDenom {
       long val;
public:
       NegativeDenom(char *message, long value)
          : ZeroDenom(message), val(value) { }
       void print () const
       { cout << msg << val << "\n\n"; }
} ;

inline void inverse(long value, double& answer)
           throw (ZeroDenom, NegativeDenom)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
         throw ZeroDenom(MSG::msg(1));
    if (value < 0)
         throw NegativeDenom(MSG::msg(2), value); }

inline void fraction (long numer, long denom, double& result)
           throw (ZeroDenom, NegativeDenom)
{ inverse(denom, result);                              // result = 1.0 / denom
  result = numer * result; }                           // result = numer / denom

int main()
{
  while (true)
  { long numer, denom;  double ans;                    // numerator/denominator
    cout << MSG::msg(3) << MSG::msg(4);                // prompt user for data
    if ((cin >> numer >> denom) == 0) break;           // enter data
    try {
      fraction(numer,denom,ans);                       // compute answer
      cout << MSG::msg(5) << ans <<"\n\n"; }           // valid answer
    catch (const ZeroDenom &zd)                        // zero denominator
    { zd.print(); }
    catch (const NegativeDenom &nd)                    // negative value
    { nd.print(); } }                                  // end of loop
  return 0;
  }
```

函数inverse()和fraction()按程序18-7中同样的方法声明异常。但是，它是函数inverse()而不是知道每个异常产生了哪些消息的异常类ZeroDenom和NegativeDenom。

程序的输出实例如图18-5所示。所使用的输入数据与该程序前面版本中的大致相同。

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -42

Negative denominator: Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:      -0.25

Enter numerator and positive
denominator (any letter to quit): exit
```

图18-5 程序18-8的输出结果

可以看出，程序的输出结果是不正确的。当分母是负数时，程序打印了合适的错误信息，但并未显示这个负数分母的值，而是要求用户输入下一个数据。怎么回事呢？

回忆一下，异常可在两种情况下抛出：在try语句块中或在try语句块之外。当异常在try语句块之外抛出时，函数立即终止，并在该函数的调用者空间中进行反复测试：可能在try语句块中也可能在try语句块之外调用抛出异常的函数。

例如，inverse()函数在try语句块之外抛出异常。当其中某个异常抛出时，inverse()函数立即终止，控制被传送给该函数的调用者fraction()。在fraction()函数中，对抛出异常的inverse()函数的调用也是在try语句块之外。因此fraction()函数也立即终止，并将控制传送给main()函数。

当异常在try语句块中抛出时，控制转移到包容throw语句的try语句块结束处。try语句块后必须跟着一个或几个catch语句块，这些catch语句块的参数将被一个接一个严格地检查。如果没有找到与抛出异常相匹配的参数，就像在try语句块之外抛出异常一样处理，函数立即终止，控制传送给函数的调用者。如果找到了匹配的参数，将停止搜索，控制转移给相匹配的catch语句块。在这个catch语句块终止后，它后面所有的catch语句块（如果有的话）都将被忽略，而继续执行所有的catch语句块后的语句。

如果catch语句块的参数类型与抛出的异常类型相同时，表示异常找到了匹配的catch语句块。如果抛出的对象是从catch的参数类型派生而来，或者抛出的对象指向一个派生类的对象而catch的参数类型指向一个基类的对象，那么这个异常与catch语句块也是相匹配的。很复杂吗？正如我们在第15章中所讨论的：一个派生类对象可以代替其基类对象使用。

在程序18-8中，当处理NegativeDenom异常时，inverse()函数和fraction()函数终止，因为它们没有try语句块。当fraction()函数终止时，它将（从inverse()函数中接收来的）异常传递给main()函数。由于main()函数是在try语句块中调用fraction()函数，catch语句块一个接一个地检查。首先检查带有ZeroDenom类参数的catch语句块。由于fraction()函数抛出的NegativeDenom对象可代替ZeroDenom对象使用，因此搜索停止，执行带有ZeroDenom参数的catch语句块。它将基类ZeroDenom::print()消息发送给参数对象，并只打印消息，但不打印NegativeDenom

对象的值。因为这个值是派生类的数据成员，ZeroDenom::print()不知如何处理。

一些编译程序可能会对这个问题产生警告，但是没有任何编译程序会标识这个设计为语法错误，因为程序员有权按他认为合适的顺序排列catch语句块。

补救措施非常简单，将带有基类对象参数的catch语句块不放在前面，而放到catch语句块系列的最后。将程序18-8中的main()函数做相应修改后如下所示，这样就没有前面所出现的问题了。

```
int main()
{ while (true)
  { long numer, denom;  double ans;
    cout << MSG::msg(3) << MSG::msg(4);              // prompt user for data
    if ((cin >> numer >> denom) == 0) break;         // enter data
    try {
      fraction(numer,denom,ans);                     // compute answer
      cout << MSG::msg(5) << ans <<"\n\n"; }         // valid answer
    catch (const NegativeDenom &nd)                  // derived class
    { nd.print(); }
    catch (const ZeroDenom &zd)                      // base class
    { zd.print(); } }                                // end of loop
  return 0; }
```

18.3.3 标准异常库

C++标准库定义了由继承层次组成的几个标准的异常类。其中最重要的类是作为层次基类的exception类（全部小写）和从exception类派生而来的bad_alloc类。

exception类定义在头文件< exception>、< except.h>或< exception.h>中，exception类有一个返回一个字符指针的虚函数what()，它与上面程序18-7中的NegativeDenom类的getMsg()方法相似。没有定义字符串的内容，但我们可以设计一个继承exception类的类，然后在这个派生类中重定义what()函数。

```
class NegativeDenom {
    long val;                                  // private data for exception info
    char* msg;
public:
    NegativeDenom(long value)                  // conversion constructor
          : val(value), msg(MSG::msg(2)) { }
    const char* what() const                   // can return an arbitrary string
      { return msg; }
    long getVal() const
      { return val; }
} ;
```

bad_alloc类定义在头文件<new>或<new.h>中。当new操作不能在堆中分配所需的内存时抛出bad_alloc类的对象，但并不是所有的编译程序都支持这个异常。下面的小例子创建了一个很长的内存块链表，它使用了bad_alloc异常，并测试new操作是否返回空指针。

```
#include <iostream>                            // include files
#include <exception>
#include <new>
using namespace std;

struct Block
```

```
{ char a[1000];                                    // memory block
  Block* next;
Block (Block* ptr)                                 // hook up before ptr
  { next = ptr; } } ;

int main()
{ Block *list = 0, *p;  int cnt = 1;
  while (true)                                     // go until it crashes
{ try {
    p = new Block(list); }                                  // this can fail
  catch (bad_alloc &bad)
  { cout << bad.what() << endl;                             // message as recovery
    exit(0); }
  if (p == 0)                                               // message as recovery
    { cout << "Out of memory\n\n";  exit(0); }
  list = p;                                                 // success: top of list
  if (++cnt%100 == 0)
    cout << "Block #" << cnt << endl; }                     // watch progress
while (p != 0)
 { p = p->next; delete list; list = p; }                    // return memory
return 0; }
```

异常处理机制并不支持中断这样的异步异常，而是处理顺序执行过程中发生的同步异常，如溢出、越界错误、资源分配错误、错误的输入数据等。在执行流中正确的情况下不能使用异常，例如，终止正常处理的某个阶段（列表迭代的结尾）并始另一个阶段。否则会使程序无必要地更慢和更复杂。

使用C++异常主要有两大好处。其一是为发现错误的地方与处理错误的地方提供了通信。另一好处是在终止被调用的函数过程中会返回栈，并安全地将控制转移给调用者。如果被调用的函数在栈中分配了对象、将调用对象的析构函数，就像这些函数正常返回一样。这样保证了系统资源的正常返回，并防止出现死锁和资源的枯竭。

18.4 类型转换运算符

本节的内容实际上并不属于这一章，但不能在本书前面对它们进行讨论，因为要基于继承、模板、异常等的高级概念。

事实上，我们也犹豫过是否要讨论类型转换运算符。因为它们对于C++语言来说比较新，而且在业界中的使用也相当有限。没有确凿的证据说明这些运算符比我们在此之前常用的、标准简单的类型转换要好。

但是，这些转换运算符代表了一些有趣的软件工程思想，完全值得我们去熟悉它们。至于是否在实践中使用它们，则由程序员自己决定。

正如我们在前面所讨论的，转换运算符和转换构造函数削弱了C++中的强类型规则，它们使得类型之间的转换成为可能。客户端代码程序员和维护人员可能困惑于可行的转换和实际发生的转换。

为了帮助程序员应付这种情况，C++中引进了一些其他的转换运算符，这些转换运算符比前面所讨论的标准转换运算符要累赘得多。实际上，这也是它们的一个优点，因为在源程序中认出这些转换运算符要比标准转换容易。

18.4.1 static_cast运算符

在可以使用标准转换运算符的地方都可以使用static_cast运算符。这样说有些夸张。

准确地说应该是适用于任何标准转换有意义的地方，但在使用标准转换运算符比较危险的地方不能使用static_cast。稍后我们会讨论几个例子。

static_cast是一个一元运算符，用某个类型的操作数接收另一个类型的数据。程序员必须在括号（ ）中指定操作数（要转换的类型的对象或类型表达式）。此外，还要在尖括号< >中指定目标类型，与模板语法类似。

```
valueOfTargetType = static_cast<TargetType>(valueOfSourceType);
```

可以看出，这个转换实际上不是一元运算，因为它既需要源类型的数据值（一个操作数），也需要目标类型名（另一个操作数）。但它又不是一个二元运算，因为转换运算符并不像一般的二元运算符那样出现在两个操作数之间。

该转换运算符并不只限用于上面例子中的赋值语句中，任何能使用目标类型TargetType（假设已定义）数据的地方都可以使用这个运算符。下面是一个简单例子：

```
double d;  int i = 20;
d = static_cast<double>(i);              // ok: d is 20.0
```

可能有人会问为什么这样做比以前的可靠的double类型转换好。它们的功能是完全一样的。

```
double d;  int i = 20;
d = double(i);                    // ok: d is 20.0
```

下面是一个比较复杂的例子。Account类提供了几个转换运算符用于获取其成员的值。为简单起见，我们使用固定大小的数组存放所有者姓名。

```
class Account {                                          // base class of hierarchy
protected:
  double balance;                                        // protected data
  int pin;                                               // identification number
  char owner[40];
public:
  Account(const char* name, int id, double bal)          // general
  { strcpy(owner, name);                                 // initialize data fields
      balance = bal; pin = id;}
  operator double () const                               // common for both accounts
  { return balance; }
  operator int () const
  { return pin; }
  operator const char* () const
  { return owner; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                                 // increment unconditionally
} ;
```

正如我们在第15章中所介绍的，这些重载的运算符函数可使用像标准的转换运算符一样的语法被调用。

```
Account a1("Jones",1122,5000);           // create object
int pin = (int)a1;
double bal = (double) a1;                // legitimate casts
const char *c = (const char*) a1;
```

在这种情况下，也可以使用static_cast运算符，其功能与标准转换运算符完全相同。

```
Account a1("Jones",1122,5000);          // create object
int pin = static_cast<int>(a1);         // ok
double bal = static_cast<double>(a1);
const char *c = static_cast<const char*>(a1);
```

注意，这些static_cast运算符能正常使用的原因只有一个：Account类支持重载运算符int、double及const char*。否则，对Account对象使用static_cast运算符与使用标准转换运算符一样是无用的。

标准转换运算符和static_cast之间的主要区别在于对指针的转换上。标准转换运算符的使用依赖于程序员的经验。如果我们将一个double指针指向一个int类型的变量，这意味着我们有很好的理由这样做，没有人会告诉我们应该做什么和不应该做什么。

程序18-9演示了一个指针转换的简单例子。程序的执行结果如图18-6所示。

程序18-9 使用标准转换运算符对指针进行转换的例子

```
#include <iostream>
using namespace std;

class Account {                                    // base class of hierarchy
protected:
  double balance;                                  // protected data
  int pin;  // identification number
  char owner[40];
public:
  Account(const char* name, int id, double bal)    // general
  { strcpy(owner, name);                           // initialize data fields
    balance = bal; pin = id; }
  operator double () const                         // common for both accounts
  { return balance; }
  operator int () const
  { return pin; }
  operator const char* () const
  { return owner; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                           // increment unconditionally
} ;
int main()
{
  double *pd, d=20.0; int i = 20, *pi = &i;
  pd = (double*) pi;
  cout << "i=" << *pd << "  i=" << *pi <<endl;
  Account a1("Jones",1122,5000);                   // create objects
  pd = (double*)(&a1);
  cout << "balance = " << *pd << endl;
  *pd = 10000;                                     // change data member
  cout << "balance = " << *pd << endl;
  return 0;
  }
```

在main()开始处，两个指针pd和pi都指向一个整数变量i，后来将这两个指针进行间接引用以打印i的值。如我们所看到的，整数指针pi正确地获取到了i的值，而用double类型的指针pd时出现问题。

```
i=9.88131e-323  i=20
balance = 5000
balance = 10000
```

图18-6　程序18-9的输出结果

然后，将double类型的指针pd指向一个Account对象a1，将这个指针进行间接引用，程序不仅获取了对象的数据成员balance的值，而且还能对它进行任意修改。

在这里，如果使用static_cast，则其行为与标准转换运算符大不一样。整数地址可作为（double*）转换的操作数，因而double类型的指针能表示变量i的值，尽管这个表示是错误的。但如果使用static_cast运算符，则会出现语法错误。

```
pd = (double*) pi;                          // ok
pd = static_cast<double*> (pi);             // syntax error
```

类似地，double类型的指针pd可以访问并修改Account对象的数据成员，这只是因为Account地址可作为标准转换运算符的操作数。但如果使用static_cast，则同样会出现语法错误。

```
Account a1("Jones",1122,5000);              // create object
pd = (double*)(&a1);                        // ok
*pd = 10000;                                // ok
pd = static_cast<double*>(&a1);             // syntax error
```

这并不意味着static_cast运算符不能用于转换指针，它可以对指针进行转换。但是，当从软件工程角度而言转换没有意义时，不能用它转换指针。当指针转换有意义时，即使是不安全的转换，也可以使用static_cast。例如，SavingsAccount类，它是从Account类公共派生而来的。

```
class SavingsAccount : public Account {
       double rate;                                 // fixed interest rate
public:
       SavingsAccount(const char* name, int id, double bal)
         : Account(name, id, bal), rate (6.0) { }
       void payInterest()                           // pay once a month
       { balance += balance * rate / 12 / 100; }
} ;
```

SavingsAccount对象不仅具有Account对象所具有的功能，而且还增加了数据成员和成员函数。因此，一个Account指针可以毫无困难地指向一个SavingsAccount对象，这是安全的，而且不需要任何标准或非标准的转换

```
Account a1("Jones",1122,5000);              // create objects
SavingsAccount a2("Smith",1133,3000);
Account *pa = &a2;                          // save conversion, no cast is needed
```

一个SavingsAccount指针不能指向一个Account对象，因为该指针可以发送一个基类对象不能响应的消息给Account对象。

```
SavingsAccount *psa = pa;                   // syntax error
```

当然，如果Account指针实际上指向一个SavingsAccount对象时，这种赋值（转换）

是有意义的，但必须用显式的转换告诉编译程序。可以使用标准转换运算符。

```
psa = (SavingsAccount *)pa;                    // explicit cast is ok
```

在这种情况下，可使用static_cast运算符转换指针，以代替标准转换。

```
psa = static_cast<SavingsAccount*>(pa);  // this is perfectly ok
```

当然，static_cast运算符可用于非安全的转换。例如，我们将SavingsAccount指针指向一个Account对象，static_cast运算符和标准的转换运算符一样不会提出异议。

```
psa = static_cast<SavingsAccount*>(&a1);  // this is perfectly ok
```

总而言之，static_cast运算符与标准转换运算符哪个更好，答案要从两方面来考虑。首先，static_cast运算符比较累赘：在程序代码中易于识别它。其次，它使用时比标准转换运算符更为严格。如果有人认为这些优点一定程度上被要求多输入文字的缺点抵销了，这种观点应予更正：在许多场合，C++的创建者Bjarne Stroustrup说过，使用类型转换越少越好，任何阻止我们使用转换的原因都是有利的。

18.4.2 reinterpret_cast运算符

不像static_cast运算符那样有限制，reinterpret_cast运算符可用于任何能使用标准转换运算符的地方。

当编译程序不知道指针所指向的实际类型时，可使用reinterpret_cast运算符。在下面的例子中，整数指针p指向一个double类型的值。在最后一行，将p的值赋给了double类型的指针。编译程序不知道指针p实际上指向一个double类型的值，程序员使用reinterpret_cast运算符告诉编译程序。

```
double y = 42;
int *p = reinterpret_cast<int*>(&y);           // potential trouble
double *q = reinterpret_cast<double*>(p);      // p points to double
cout << "The answer is " << *q << endl;        // it prints 42!
```

使用标准转换运算符int*和double*也能达到同样的效果。

```
double y = 42;
int *p = (int*)&y;                             // integer p points to double: trouble
double *q = (double*)(p);                      // ok because p points to double
cout << "The answer is " << *q << endl;        // it prints 42!
```

由于reinterpret_cast运算符更为明显，所以我们认为它比标准转换运算符要好。

注意，这里不能使用static_cast运算符：它可以转换不同类型的数据，但不能转换指针。还要注意static_cast运算符是可移植的，因为编译程序要检查类型是否相关或是否有合适的转换运算符或转换构造函数。

但reinterpret_cast运算符不能保证可移植，它只是提取源表达式中的位集合，然后根据目标类型的规则进行解释，而不能保证在不同机器上有同样的结果。其结果实际上依赖于具体的机器。

尽可能少地使用reinterpret_cast运算符。然而，当必须使用转换运算符时，最好使用reinterpret_cast而不是标准的转换运算符。

18.4.3 const_cast运算符

const_cast运算符有能力消除常量数据或常量对象的常量特性。其语法与C++中其他的

现代转换运算符相同，在尖括号< >中是目标类型，在括号()中是源表达式（必须有括号）。

```
nonConstValue = const_cast<TypeName>(constValue);
```

其语法和语义比其他的转换运算符更为严格。它所做的是去掉源值constValue的const特性，因而可将一个常量值constValue赋给一个非常量值nonConstValue。要求nonConstValue的类型一定是TypeName，constValue的类型一定是const TypeName。

考虑下面的简单例子，由于我们把变量d定义为const，一般的指针不能指向它（以避免通过间接引用该指针来改变值）。

```
const double d = 42;
double *pd = &d;          // error: to prevent *pd = 21
```

指向一个常量的指针可以指向变量d，但不能用该指针改变目标的值。

```
const double d = 42;
const double *pd = &d;    // ok but not very useful
*pd = 21;                 // syntax error: a pointer to const
```

这样，可使用const_cast运算符去掉常量属性，使定义为const的数据值可改变。

```
const double d = 42;
double *pd = const_cast<double*>(&d);          // remove const
*pd = 21;                                      // now it is ok
cout << "The answer is " << *pd << endl;       // it prints 21
```

这个技巧甚至连标准的C类型转换也实现不了。显然，不应该经常使用这个运算符。一个可能必须使用它的情况是，维护的代码中某个变量被定义为const，但新的条件要求改变这个变量。与其修改已存在的定义，不如增加新的代码，使用const_cast运算符改变这个变量。

使用const_cast运算符去掉了保护机制。指针可以不必指向一个常量，当改变该指针所指向的对象时，可对这个指针进行间接引用。这是个非常危险的方法。

再来考虑一下Account类。

```
class Account {                                    // base class of hierarchy
protected:
  double balance;                                  // protected data
  int pin;                                         // identification number
  char owner[40];
public:
  Account(const char* name, int id, double bal)    // general
  { strcpy(owner, name);                           // initialize data fields
    balance = bal; pin = id; }
  operator double () const                         // common for both accounts
  { return balance; }
  operator int () const
  { return pin; }
  operator const char* () const
  { return owner; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                           // increment unconditionally
};
```

如果试图对一个Account类的const对象调用一个非const的成员函数（如operator+=()），编译程序将拒绝接受这样的代码。

```
const Account a1("Jones",1122,5000.0);     // create object
a1 += 1000.0;                              // syntax error
```

如果将一般的Account指针指向一个const对象，编译程序同样会拒绝接受这样的代码，以免通过间接引用指针来改变对象的值。

```
const Account a1("Jones",1122,5000.0);     // create object
Account *pa = &a1;                         // syntax error
```

如果指针指向一个常量对象，该指针必须定义为指向const对象的指针。这是允许的，但不能使用这个指针来改变对象的状态。

```
const Account a1("Jones",1122,5000.0);     // create object
const Account *pa = &a1;                   // ok
*pa += 1000.0;                             // syntax error
```

然而，我们使用const_cast运算符可以将一般的指针指向一个const对象。

```
const Account a1("Jones",1122,5000.0);       // create object
Account *pa = const_cast<Account*>(&a1);     // ok
*pa += 1000.0;                               // this is permitted
```

结果，常量对象的状态改变了——它的收支情况现在为6 000美元，但并未对该对象进行直接的显式操作。

const_cast运算符的惟一任务是去掉const的保护，它不能处理其他任何类型转换。如果nonConstValue（转换结果）与constValue（被转换的值）不是同一类型，则需要进行类型转换，而且这个类型转换是另外一个单独的额外步骤。

例如，Account类的const char*()转换运算符返回一个字符指针，不能（也不应该）用该返回的指针改变Account对象内的字符数组内容。

```
const Account a1("Jones",1122,5000.0);     // create object
char *c2 = static_cast<const char*>(a1);   // syntax error
```

只能用指向常量的指针对这个指针赋值，而且不能用这个指针改变Account对象的状态。

```
const Account a1("Jones",1122,5000.0);           // create object
const char *c2 = static_cast<const char*>(a1);   // this is ok
strcpy(c2,"Jones");                              // syntax error
```

对Account对象使用const_cast运算符不会有帮助，因为目标值和源值是不同的类型。

```
const Account a1("Jones",1122,5000.0);           // create object
char *c2 = const_cast<char*>(a1);                // syntax error
```

由于const_cast运算符只能去掉const特性，因而完全可以先将常量Account对象转换为指向常量的指针（使用static_cast或标准转换运算符），然后将指向常量的指针转换为一般的指针（使用const_cast）。

```
const Account a1("Jones",1122,5000.0);                 // create object
const char *c1 = static_cast<const char*>(a1);         // this is ok
char *c2 = const_cast<char*>(c1);                      // and this is ok
strcpy(c2,"Jones");                                    // not a syntax error
```

这样，没有对常量Account对象进行显示处理，就改变了拥有者的姓名。

18.4.4 dynamic_cast运算符

dynamic_cast运算符是C++支持运行时类型信息（run_time_type information，RTTI）的一个组成元素。该集合中的其他元素还有typeid运算符和type_info结构。

dynamic_cast运算符用来将基类的指针（或引用）转换为某个派生类的指针（或引用）。正如我们在前面所讨论的，static_cast运算符（或标准类型转换）也可以这样用，但程序必须知道对象的类型，以确保将指针转换为正确的类。

dynamic_cast运算符的语法与其他的转换运算符相同：参数指针（或引用）放在括号中、参数要转换的目标类型放在尖括号中。如果参数指针确实属于要求的目标类型，运算符不改变地返回指向该对象的参数指针。如果参数指针指向目标类型的一个派生类对象（直接或间接派生），也将返回指向对象的指针。否则，返回空，程序可以检查这个值。

这种方法要求类的继承层次中既含有虚函数，又含有非虚函数。如果继承中没有虚函数，则不能使用这种方法。

例如，简化了的Account类有一个虚函数display()，下面显示了Account对象的内容。

```
class Account {                                          // base class of hierarchy
protected:
  double balance;                                        // protected data
  int pin;                                               // identification number
  char owner[40];
public:
  Account(const char* name, int id, double bal)          // general
  { strcpy(owner, name);                                 // initialize data fields
    balance = bal; pin = id;}
  virtual void display()                                 // virtual function for RTTI
  { cout.setf(ios::fixed,ios::floatfield); cout.precision(2);
    cout <<setw(6) << pin << setw(20) << balance
         << '  " << owner <<endl; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                                 // increment unconditionally
} ;
```

派生类SavingsAccount增加了一个payInterest()方法，并重新定义了基类方法display()，这样可以显示所增加的数据成员interest。

```
class SavingsAccount : public Account {
       double rate, interest;                  // accumulated interest
public:
  SavingsAccount(const char* name, int id, double bal)
     : Account(name, id, bal), rate (6.0), interest(0) { }
  void payInterest()                           // pay once a month
  { double pay = balance * rate / 12 / 100;
    balance += pay;  interest += pay; }
  virtual void display()
  { cout.setf(ios::fixed,ios::floatfield); cout.precision(2);
    cout <<setw(6) << pin << setw(8) << interest << setw(12)
         << balance << "   " << owner << endl; }
} ;
```

在这里，我们定义了两个对象：一个为基类对象，另一个为派生类对象。我们还定义了

一个Account指针，并使用了dynamic_cast运算符，首先将这个指针指向基类对象，然后再指向派生类对象。正如使用dynamic_cast运算符的通常做法，我们检查指针是否为空：如果不为空，答案是肯定的，由这个指针指向的对象可用作dynamic_cast运算符中指定的类的对象；如果为空，答案是否定的。

```
Account a1("Jones",1122,5000);                    // create objects
SavingsAccount a2("Smith",1133,3000);
Account *pa = dynamic_cast<Account *>(&a1);       // ok
if (pa == 0)
   cout << "Null pointer\n";
else
   pa->display();                                 // Jones
pa = dynamic_cast<SavingsAccount *>(&a2);         // ok
   if (pa == 0) cout << "Null pointer\n";
else
   pa->display();                                 // Smith
```

在这个例子中，对于这个问题指针是否为空并不重要，因为作为赋值目标的指针是一个基类指针——无论是指向基类对象还是派生类对象，它都不会造成破坏。的确，第一个转换返回指向对象a1的指针，第二个转换返回指向对象a2的指针，该指针被转换为指向基类指针。由于display()函数是多态的，它在第一种情况下以基类的格式显示数据，在第二种情况下以派生类格式显示数据。这段程序代码演示了在指针指向的对象类型与运算符中指定的类型相同时dynamic_cast运算符的行为。

在下面的程序段中，我们再次使用基类的指针作为目标。首先，我们将它指向基类对象，并询问它是否能完成Account类的操作。结果是它能处理，即运算符返回指向对象的基类指针。尽管如此，display()消息以派生类的格式显示该指针所指向的对象，因为这个函数是虚函数，而且对象属于派生类。接着，我们检查对象a1能否完成SavingsAccount对象的操作。结果不能，a1是一个基类对象，操作符返回空。

```
pa = dynamic_cast<Account *>(&a2);                // ok
if (pa == 0)
   cout << "Null pointer\n";
else
   pa->display();                                 // Smith
pa = dynamic_cast<SavingsAccount *>(&a1);         // null
   if (pa == 0) cout << "Null pointer\n";
else
   pa->display();
```

下面的程序段更为有趣，因为它使用了派生类指针。首先，我们检查对象a1能否完成派生类的操作。与前面一样，答案是不能。赋值的目标是基类指针（正如上面的例子）还是派生类指针（这里的情况）并不重要，因为都将返回空。接着，我们检查对象a2能否完成派生类的操作，答案与第一段代码一样：可以完成。在第一段代码中，我们将结果转换为基类指针，因此只能调用Account方法及派生类中的虚方法。然而在这里没有转换，赋值的目标是一个派生类指针，它能访问基类函数、虚函数、也能访问派生类中定义的函数。

```
SavingsAccount *psa = dynamic_cast<SavingsAccount *>(&a1);    // 0
if (psa == 0)
  cout << "Null pointer\n";                                   // null pointer
else
```

```
  psa->display();                                          // no display
psa = dynamic_cast<SavingsAccount *>(&a2);                 // ok
if (psa == 0)
  cout << "Null pointer\n";
else
 { psa->payInterest();                                     // derived method
   psa->display(); }                                       // Smith
```

dynamic_cast运算符提供了强有力的方法，检查某个给定的对象能否完成所要求的操作。因为它是一个相当新的语言特性，因此并不是所有的编译程序都能支持这个运算符。即使有的编译程序能支持，却也不能缺省地使用它。为了使用dynamic_cast，必须设置编译程序标志或选择显式地支持RTTI特性。

与new运算符类似，C++提供了其他的方法来检查转换是否成功：抛出一个异常。如果指针不指向运算符中指定的类的对象，则抛出bad_cast异常。这对于引用尤为重要，C++指针可能指向一个对象，也可能不指向对象；但引用总是指向一个对象——因为它们不能为空。对于引用，使用dynamic_cast时不会（像在指针中那样）进行检查或询问，但要声明实际上引用指向的是转换中指定的类的对象。当确认失败时，则抛出适当的异常。

18.4.5 typeid运算符

还有另一种决定基类指针转换成什么类的方法，其基础是使用typeid运算符。typeid运算符有两方面的功能：检查参数的类名、检查所指向的对象是否属于给定的类。

与转换运算符不同，typeid运算符的参数是一个对象，而不是指针，它返回指向库类type_info对象的引用。其调用的实现依赖于编译程序。但在该类的成员中总有一个成员函数name()。这个函数返回一个字符数组，其实现也依赖于编译程序，这个字符数组经常是运算符的参数所属的类名，或class关键字后的类名。

有些编译程序将type_info构造函数定义为私有的。在这种情况下，C++程序并不创建type_info的对象，而是发送一个消息给typeid运算符的返回值。

```
Account a1("Jones",1122,5000);                  // create objects
SavingsAccount a2("Smith",1133,3000);
const char *c1 = typeid(a1).name();             // get class name
const char *c2 = typeid(a2).name();
cout << c1 << endl;                             // prints "class Account"
cout << c2 << endl;                             // prints "class SavingsAccount"
```

当然，这只对调试有用。typeid运算符实际的功能基于以下事实：该运算符不仅用于某个类的表达式中，也可用于作为标识符的类名（无引号）。这样，我们可以对类名使用typeid运算符，以及对对象使用typeid运算符，然后将其结果进行比较，如果相等运算符返回真，表明对象属于指定的类。

```
if (typeid(Account) == typeid(a1))                 // true
   cout << "a1 is Account\n";
if (typeid(SavingsAccount) == typeid(a2))          // true
   cout << "a2 is SavingsAccount\n";
if (typeid(Account) == typeid(a2))                 // false
   cout << "a2 is Account\n";
if (typeid(SavingsAccount) == typeid(a1))          // false
   cout << "a1 is SavingsAccount\n";
```

在处理由基类指针指向的异构对象集合时，将指针作为typeid参数时必须间接引用该指针。

```
pa = &a2;
if (typeid(Account) == typeid(*pa))              // false
   cout << "pa points to Account\n";
if (typeid(SavingsAccount) == typeid(*pa))       // true
   cout << "pa points to SavingsAccount\n";
```

注意，这些比较操作并没有对对象进行比较——C++的结构没有定义比较操作，也没有对指针进行比较：不像其他的特殊转换运算符，typeid运算符返回的是一个对象，而不是指针。可将重载的比较运算符应用到typeid运算符返回的type_info对象上。

typeid运算符很有用，但又容易滥用。它是使用虚函数的非结构化竞争对手。千万不要过多地使用它。

18.5 小结

本章所讨论的内容都比较新。并不是所有的编译程序和类库都能支持它们。业界也没有积累太多使用它们的经验。因此，要谨慎地使用C++的这些特性。

异常的代价是占用内存和延长执行时间，这对于某些应用程序很重要。对于大部分应用程序而言，这不是很重要。重要的是在程序中如何使用异常去构造并简化应用程序的控制流。

似乎抛出内部数据类型的异常不是很有用，因为异常处理机制不能区别源代码中不同地方抛出的同一类型的值。更有用、更引人注意的是预设的类将错误发现处的异常信息传递到错误恢复的地方，虽然这样会增加应用程序中类的个数及要编写的代码数。

许多人认为异常处理机制简化了应用程序的控制流，允许设计人员将主流处理代码与异常处理代码隔离开来。这种想法可能是正确的。使用if和switch语句的确很让人困惑，使得源代码更复杂难懂。但这种复杂性反映了程序所要处理的任务的复杂性，不应该责怪复杂代码所使用的控制结构。使用标准的控制结构处理不同情况的优点是：所有的代码都在同一个地方，没有被拆分。

当使用异常时，维护人员要完成额外的任务，即对设计人员所做的拆分处理的决定进行分析。这些决定常常比较复杂，且有很大的随意性，本应在一起的处理有时被拆分成几个部分。这就增加了程序的复杂性。

我们认为只需在下面的情况中使用异常——处理错误的程序代码所需的信息只有发现错误的程序部分才知道。在这种情况下，使用异常将程序一个部分的必要信息传递给程序另一部分。这很好，但是要记住，需要将程序一个部分的信息传递给另一部分的起因是，以前的设计决定将处理分开成几个部分。重新考虑这个设计决定可能可以消除额外的异常处理的需要。

在使用异常时，要让代码易于维护。应声明每个函数抛出的所有异常或从函数的服务器传递来的异常，不要声明函数不能抛出的异常。尽可能地在源代码和单独的文件中对异常处理文档化。即使测试异常并不容易，也要保证对每个异常处理机制进行测试。

本章最后所讨论的类型转换运算符非常新，业界在这方面也没有很多经验。必须承认本书的作者也没有使用它的很多经验——使用C模式的标准转换就觉得足够了。

是的，标准的转换很容易遭到滥用，它也允许我们在指针之间进行无意义的转换，

static_cast运算符可以防止我们这样做。如果一定要进行无意义的转换，可使用reinterpret_cast运算符，它允许我们像使用标准转换那么容易。

然而，许多专家认为这些转换运算符增加了程序的可读性：它们易于识别而且容易吸引维护人员的注意力。这的确是真的，我们建议大家使用这些转换运算符。有了更多的使用经验后，也许就会喜欢它们。

第19章 总 结

至此，我们已经用了很多时间和精力完成这本书。但是必须承认，这本书的工作量这么大，能够写到最后这一章实在是很不容易的。现在，我们回顾一下本书的内容。

本章，我们不再学习新的语法，而对C++语言强大的、精彩的、令人困惑而又甚至不安全的基本特点进行总结。我们已经掌握了整个主题，理解了如何将不同的组件恰当地组装在一起，并且掌握了如何在设计中加入自己的想法，以及在使用C++时，哪些地方应小心。

在前面的章节中，我们限制了讨论的内容，因为有些讨论涉及的知识大家都还不熟悉。现在我们已经讨论了所有的内容，这些限制不再存在了，因此本章很值得一读。

C++语言是创建大型计算机程序的软件工程语言。它追求的目标很多，这些目标有时是相互冲突的。一方面，C++试图成为面向性能的系统程序设计语言：它提供了低级的操作（如移位和按位逻辑运算符），并允许访问机器资源（如寄存器、易变数据类型、基于指针的算术操作）。另一方面，C++还试图帮助将程序分成几个小的独立模块，可以由不同的程序员开发不同的模块，并使他们之间的通信尽可能少。

C++语言具有以下特点：

- 是可读性强的高级语言（数据聚集、控制流、名字作用域）。
- 是快速敏捷性的语言（独一无二的速记（shorthand）运算符、简洁的表达式）。
- 使用字符串和动态内存管理。
- 使用提供的库（事实上的标准）。

19.1 作为传统程序设计语言的C++

与许多高级语言不同，C++语言区分大小写；与许多现代高级语言相似，C++语言忽略空格的存在(有两三个例外)。它使用行尾（end-of-line）注释，但不使用嵌套的块注释。

与大多数其他的程序设计语言相似，C++语言提供了内部数据类型及其操作，但它所提供的内部数据类型非常有限——只有简单的整数和浮点数值。

19.1.1 C++内部数据类型

为了获得最优的性能，C++提供的整数类型在任何平台上的速度都是最快的类型。整数的长度依赖于机器：在16位机器中其长度为16位，在32位机器中其长度为32位。这将影响程序的可移植性。在C++语言中这是非常典型的，不能保证同一程序在不同机器上的运行结果完全相同。

为了增加复杂计算的灵活性（即在可能时节约内存）和计算能力（即在需要的时候扩展值域），C++语言为精确地使用内存提供了类型修饰符（`short`、`long`、`unsigned`）。C++没有对不同类型的大小标准化。它只是要求一个`short`类型的值不能比整数类型的值长；也要求一个长整数类型的值不能比一个整数类型的值短。

因此，在现代计算机上，`short`类型值长度通常为16位，`long`值经常为32位。想获得可

移植性的程序员应避免使用简单的整数，而应使用short或long修饰符。想获得速度的程序员要使用integer而避免使用short或long修饰符。

使用unsigned的值可以更加精细地使用内存，但是这也是一个颇有争议的问题。一方面，定义unsigned值就是告诉维护人员该值本质上是正数而不能为负数。使用unsigned修饰符后所能表达的最大整数值是给定结构（同样数量的二进位）的2倍。另一方面，混用signed和unsigned类型的数据可能导致计算不正确的答案。为了避免这些错误，许多程序员放弃无符号数类型的潜在优点，而不使用它。

为了简化程序员的选择，C++支持缺省情况　如果程序员没有指定数据是signed或unsigned，则缺省认为是signed　如果没有指定数据是short整数或long整数或整数，则缺省认为是整数。

为了获得最优的性能，C++对计算结果的上溢出和下溢出都不做测试。程序中应该测试的每个问题都应该在程序的源代码中显式地测试。如果程序不想花时间检查结果的合法性，C++不会提供任何缺省的测试或者警告。

C++语言将字符作为另一种整数看待。一个字符占一个字节或两个字节（扩展字符集）。对字符实施算术运算在C++中是合法的。在程序中常常广泛地使用它们，但不同机器使用不同字符集时会产生可移植性问题。

C++语言允许程序员定义signed或unsigned字符。对缺省类型没有任何标准，因此，在一些机器上为unsigned的值在另一些机器上可能为signed。认为字符不能包括负值是个好的想法，如果可能出现负值（例如行尾代码）时就要使用整数代替字符。

字符文字包含在单引号中，不应该将它们和用双引号括起来的字符串文字互相混淆。C++没有将字符串的长度和字符串的内容存储在一起，而是使用0代码标志字符串的结束。因此，字符串文字的长度比文字中字母的个数要多一个。

C++语言支持三种不同大小的浮点数类型：float、double及long double，其大小分别为4个字节、8个字节和10个字节，其精度分别为7位、15位和19位。这些属性是依赖于机器的。C++的浮点常量通常是double的，而不是float或者long double。在大多数情况下，这并不重要。如果需要指定文字是浮点数，应该使用合适的后缀。C++提供固定小数点的表示法和（有指数的）科学表示法。

布尔类型有两个值：true和false。它们也被看做小的整数。bool类型的布尔值大小是一个字节而不是一位。C++没有将布尔值压缩为一位，是因为在C++中访问位需要逻辑操作和移位。在空间效率和时间效率之间，C++选择了时间效率，因为字节是内存的最小单位，可以直接寻址。

可以使用预处理器#define指示符定义任何内部数据类型的字面值的符号名。预处理器将用实际字面值替换源代码中的每一个符号名。由于这个过程是在编译程序编译源程序之前进行的，因而预处理指示符中的错误常常很难发现。最好使用const修饰符，因为用const修饰符定义的名字遵循作用域规则（在#define中定义的名字是全局的）。

对于每一种数据类型，C++支持两种派生的数据类型：指针类型和引用类型。这两种类型都包含值的地址，但它们的使用语法不同。

C++允许在任意不同类型的的数值之间进行转换，即当需要某个类型的数据时，使用另一个类型的数据来代替。布尔类型和数值类型之间也可以互换——不会出现语法错误。对于数

值类型的数据而言，C++是一种弱类型语言。

不同类型的指针（或引用）的值不能相互转换（也不能转换成该类型的值）。对于地址而言，C++是一种强类型语言——即使不同类型的指针包含同一地址而进行转换时，也会出现语法错误。

指针（和引用）之间的转换可以显式进行，但与之相关的完整性问题是程序员的任务，如果转换结果没有合理的意义或者在不同的计算机体系结构之间无法移植，编译程序都不会产生语法错误。

19.1.2 C++表达式

C++有一组基于数值类型的常规运算符，例如，符号运算符、算术运算符、关系运算符、相等运算符及逻辑运算符，但没有指数运算符。与大多数其他的程序设计语言相似，没有隐式的乘法运算——必须使用*作为显式的乘法运算符。

C++将语句作为表达式看待。为了统一，C++将赋值和冒号也作为运算符（尽管它们的优先级最低）。因此，C++编译程序会把错误的构造作为合法的代码接受。

由于运算符很多，C++还使用两个符号的运算符甚至三个符号的运算符。在C++中，一个运算符（与一个关键字）的意义常常为了不同的目的被重用，这样，其实际意义依赖于上下文。

由于内部数据类型的大小依赖于机器，C++允许程序员计算某个给定变量的大小（给定变量名）或计算某个给定类型的任意变量的大小（给定类型名）。

逻辑运算符、关系运算符和相等运算符都返回布尔类型的值“true”或“false”，这些布尔值可以被灵活地转换为数值类型的1（真）和0（假）。而且，当需要使用一个布尔值时，可使用任何数值类型的数据来代表，不会出现语法错误。0可以被转换为false，任意非0值可以被转换为true。这样常常强迫编译程序接受语义错误的代码。

另一个错误的根源是相等运算符==，它被写作两个连续的等于符号，如果漏掉一个=不会产生语法错误，但完全改变了源代码的意义。这常常会浪费时间，并且令人沮丧和焦虑。

逻辑运算符&&和||的优先级不同，&&比||具有更强的绑定性，因此可以省略额外的括号。这两个逻辑运算符都是短路（short-circuit）运算符，在复合逻辑表达式中，首先计算第一个操作数，如果从第一个操作符的结果就可以知道整个操作的结果，则不需要再计算第二个操作数。

C++有一些独特的运算符，它们可以访问底层的计算机内存中的信息表示。按位逻辑操作符是这种类型的操作符，包括按位或运算符、按位异或运算符、求补（求反）运算符。它们分别对操作数的每一位进行操作，逐位地产生结果。

位移操作将给定的位模式向左或向右移动。当向左移或者将一个正数值向右移动时，插入0，这些操作是可移植的。当负数值向右移动时，结果依赖于实现：即或者插入0（逻辑移位）或者插入1（算术移位）。这个操作是不可移植的。

另外一些独特的运算符包括加1和减1运算符，它们模拟了汇编语言的类型处理，对单个左值操作数提供副作用（递增1或递减1）。这些操作符可以作为前缀运算符或者后缀运算符。前缀运算符先实施运算，然后将运算后的结果值用到其他的表达式中。后缀运算符是在其他表达式中使用后才实施运算。

C++指定了表达式中的运算符的计算顺序，但并没有指定操作数的计算次序。因此，C++程序不能依赖于表达式中某个操作数的计算次序。有副作用的操作数（加1和减1运算符）常常是引起可移植性问题的原因。最好将加1和减1运算放在单独的表达式中，避免可移植性问题。

另一个独特的运算符是条件运算符：根据第一个操作数的值，计算第二个操作数（第一个操作数为真），或者计算第三个操作数（第一个操作数为假）。

还有一组独特的运算符包括算术赋值运算符和逗号运算符。这些运算符有助于编写简明扼要的C++代码。

C++中的二元运算符的操作数类型通常是完全相同的。当源代码中给出的操作数属于不同类型时，C++使用拓宽转换，即一个较短的操作数被转换为表达式中类型最宽的操作数。在赋值运算符中，右边的操作数被转换为左边操作数的类型，这样可能会影响精确度。

19.1.3 C++控制流

与其他语言一样，C++中的语句是按顺序依次执行的。每一条语句用分号结束。可以使用语句块（复合语句），语句块是由“{”和“}”括起来的语句组成的，可以有局部变量。在语句块的闭花括号“}”后没有分号。

复合语句可以嵌套使用，它们可以作为一个函数体或作为一个控制语句体。在嵌套语句块中定义的局部变量在语句块外是不可见的。

C++有一组标准的控制结构。if-else语句不使用then关键字，当if语句中的表达式为任意类型的非零值时，执行条件为真的分支；当if语句中的表达式为零（假）时，执行条件为假的分支。

C++中实现了重复操作，支持三种形式的循环语句：while循环（允许一次也不执行），do-while循环（强制至少执行一次），和for循环（主要用于重复次数为某个固定的数）。

C++中一个常用的程序设计办法是将循环测试与赋值结合起来。这时要注意括号的使用：不小心漏掉了括号可能会改变表达式的意义，因为在C++中，比较运算符的优先级比赋值运算符要高。

C++并不支持无限制的跳转，goto语句不能离开它的作用域也不能跳出变量的定义。break语句使控制流从一个循环中跳转出来，继续执行该循环后的语句。break语句可用在这三种形式的循环语句中，常在某个条件语句中执行。continue语句忽略循环体的其余部分，返回到循环的开始处，对进一步的循环进行判断。

在C++中，switch语句支持多路分支：它根据一个整数表达式的值选择不同的执行路径（也可以使用浮点数）。如果没有找到匹配，则执行缺省情况下的语句。与其他语言不同，缺省语句是可选。如果没有缺省语句，也没有找到匹配，则执行下一条语句。为了创建多路分支结构，应在每个分支后使用break语句。

19.2 作为模块化语言的C++

与其他现代高级语言类似，C++支持为程序数据和操作创建语句块层次结构。从软件工程的角度来看，模块化对于大型工程项目，其优点主要体现在以下方面：工作分工、简化了程序设计任务、可重用及可维护的程序单元、以及可在不同级别对程序进行研究，或者从总体

上进行分析（不考虑细节），或者进行详细分析（不考虑高层问题）。

正确地使用模块化程序设计方法，不仅可以提高开发和维护方面的效率，也可以减少错误的产生。

C++支持程序员定义的聚集数据类型：数组、结构、联合及枚举类型等。这些数据类型中的成员既可以是内部数据类型的数据，也可以是其他的C++聚集类型的数据（数组、结构等）。

C++也支持程序员定义的函数。函数的层次结构对程序所要处理的现实生活中的对象行为层次进行建模。C++还支持使用标准库，标准库实现了许多通用任务。库函数是经过优化和测试的，应用范围很广。

使用库函数时，若要指定带有函数原型的头文件，则增加了使用的难度，即使这样，我们还是应掌握它。

19.2.1 C++聚集类型之一：数组

C++数组只能包含同一种类型的元素。数组最大的限制是在编译时数组的大小必须是已知的。如果数组的空间比所容纳的元素要多，则浪费了内存。如果数组的空间比所容纳的元素要少，则会导致内存混乱。

使用C++数组容易出现的另一个问题是第一个元素的下标值是0。这不能改变。这样，最后一个元素的下标值比数组的大小要小1。C++并不支持编译时下标检查。C++也不支持运算时下标有效性的判断，因为这将会影响程序的执行时间。

C++假设我们不希望在访问数组时浪费时间。当我们想对下标有效性进行检查时，可以编写自己的代码来实现；如果没有检查下标有效性，C++假设我们知道自己在做什么。对于内存很大的机器，下标错误可能不会导致不正确的运行时结果（除非内存的使用发生改变）。这是一个比较严重的问题，没有好的解决办法。

C++允许程序员通过下标或指针访问数组成员来实现数组处理的算法。前提条件是指针的加1（或减1）运算是将地址加上一个数组元素的大小，而不是加1。使用指针，可以编写出简明扼要的数组处理代码。但是使用这个方法对程序的性能没有好处。一些程序员认为这种代码很难验证。

C++支持任意维数的数组。多维数组的本质是实现为以行序为主序（row-major order）的一维数组（即右下标变动最快）。与一维数组类似，多维数组也不支持下标有效性检查。

C++将文本表示为字符数组。这些数组中必须有一个额外的元素存储用来标记数组中有效数据的结束的零岗哨值。当编译程序处理程序中的文本时，也将终止符0追加到字符串符号后，这样文本也有了额外的元素。所有处理字符数组的库函数也希望在有效数据的结束处有一个终止符0。当这些库函数改变了数组的内容时，它们将终止符0追加到有效数据的结束处，以保持字符串的有效状态。

C++既不支持数组赋值也不支持数组比较。对于任意类型的数组，程序员要确保这些赋值和比较操作正确地执行。对于文本字符串，可以使用库函数来进行赋值、比较、串接及其他的标准操作。

当字符串在内存中相互重叠时，大多数库函数不能正常工作。在写一个字符数组时，C++中没有库函数对有效空间进行检查。如果空间不足，将会导致讹用计算机内存，这是一个涉及完整性的严重问题。

19.2.2　C++聚集类型之二：结构、联合和枚举类型

C++的结构中包含了一些相关的成员。哪些成员相关哪些成员不相关，这常常是个人的判断问题。C++让程序员自己做决定，没有对成员的类型进行任何限制。

结构的定义是创建结构变量的蓝图。对于每一个结构域，程序员要提供域的类型和域名。结构的作用域定义由开花括号和闭花括号括起来，在闭花括号后有一个分号

结构变量初始化的语法与数组初始化的语法相似，即由括号括起来的一个数据值列表，两个数据值之间由逗号隔开。

使用点选择运算符可以选择一个结构对象的域（可以作为左值或者右值）。当使用指针引用一个结构变量时，点选择运算符不起作用，要使用箭头选择运算符。

C++支持同一类型的结构变量之间的赋值，也实现了值语义：将作为右值的结构变量的域逐位拷贝给作为左值的结构变量的相应域。

不允许在不同类型的结构变量之间进行赋值，即使它们有相同的复合结构，或者结构定义中的域名也相同。只有当类型名相同时才允许相互赋值。注意，使用typedef也不能使两个类型名相同；因为这只是创建了一个类型名的同义词而已。

不允许在结构变量和数值类型的变量（或指针变量、引用变量）之间赋值。因为对于程序员定义的类型，C++是一种强类型语言。编译程序对于这样的赋值将标记为语法错误。

C++不支持对结构进行比较或其他操作。要实现这些操作，必须编写程序代码来实现。

联合类型的定义与结构定义语法上类似：不同类型的域列举在作用域括号中（后接一个分号）。但这些域变量在计算机内存中不是（像结构变量那样）同时存在的。

这样可以节省程序空间：一个联合变量的信息可以是在联合定义中相互排斥的类型之一的变量信息。当然这样容易出错，因为程序员要确保从联合变量中获取的值与这个变量先前存储的值类型相同，联合本身并不保持类型信息。

如果因程序出错而获取到的值类型不同，则不会出现编译时错误，也不会出现运行时错误，只是获取到无用的位模式。为了避免这些错误，应将联合变量当做结构域使用，并在该结构中增加一个标志域来保存联合域值被初始化时的有关信息。当获取联合的值时，程序将访问这个标志域，进行相应处理。这就是在没有虚函数之前多态性的实现方式。

枚举类型定义了从一组预先定义的符号标识符中接受数据值的变量。枚举类型的定义语法与结构定义的语法相似：在作用域括号内是由逗号隔开的符号名（作用域括号后有一个分号）。这是一种比较流行的为程序定义符号常量的方式。

C++没有定义枚举类型上的操作。通常，将枚举类型作为整数来实现（从0开始），程序可能试图使用这一点，但是这样做不太好。

19.2.3　作为模块化工具的C++函数

在C++中，可以在函数中隐藏操作的复杂性。客户代码将服务器函数当做一个代码单元。这样就简化了调用的代码：客户代码以调用服务器函数的方式编写代码，而不是对数据的低级操作细节来编写代码。

对于后者，当客户代码不调用服务器函数而对数据进行处理时，维护人员必须理解语句序列所代表的意义。当使用服务器函数时，每个操作的目的都由函数名来表示（假设函数名

有足够的描述能力）。

C++函数为了实现共同目标而互相协同工作。它们协同工作的数据是相同的，可以在某个函数中对这些数据值进行设置，然后在另一个函数中使用这些数据。可以使用全局变量、参数或返回值来实现数据交换。全局变量的耦合度是隐式的，对于维护人员来说不明显，因此要尽可能少地使用全局变量。

参数的耦合度要好一些，因为它是显式的：对于维护人员（和客户端代码程序员）来说立即可以看到哪些数据值参与了函数与其客户函数之间的数据流。

在函数之间进行分工时，应该将耦合度（参数个数）最小化，这样，本属于一个整体的部分就不会被拆分在不同的函数中。若将本该属于一个整体的部分拆分在不同的函数中，则需要在这些函数之间进行通信。

在调用一个函数时，客户代码必须为函数定义中的每个形式参数提供实际参数。也可以为函数定义缺省的参数值，这样当客户代码不提供实际参数值时，也可以使用这些缺省值。

在参数传递时，C++是一种强类型语言：实际参数的个数必须与形式参数的个数相匹配，每个实际参数的类型也必须与相应的形式参数类型相匹配。如果违反了这条规则，编译程序将其标注为语法错误。

这条规则只对数值类型的数据有例外，如果在数值类型的形式参数和实际参数之间存在不匹配，则允许进行提升和转换：较小的类型的参数（enum、char、unsigned char、short）提升为整数，unsigned short提升为int或者unsigned int（根据机器的体系结构而定），float类型的参数提升为double类型。如果对实际参数进行提升后仍不能与形式参数相匹配，则使用转换：任意一个数值类型可以转换为另一数值类型，但这样有可能损失准确性（例如从double转换为整数）。

类型提升和转换不适用于程序员定义的类型、指针与引用（即使这些指针或引用指向数值类型）。类型转换只适用于数值类型。

C++语言是一种分别编译的语言。为了有助于编译，编译程序在处理某个函数调用之前就要知道函数的界面。除非在源文件中函数定义在函数调用之前（这不常见），否则应使用函数原型，即定义函数的参数类型和返回值。函数原型中的参数名很有用，但它是可选的。

一个函数只可以定义一次，根据需要可以进行多次声明（作为函数原型）。如果在几个文件中都使用了某个函数，则必须在每个文件中都分别进行声明，函数原型常常放在#include头文件中。

C++的全局函数可通过函数名和参数类型序列进行定义。当将函数定义为一个类的成员函数时，这个类名也成为函数定义的一部分。类名（如果有的话）、函数名及参数类型列表所构成的组合（即函数标识signature）必须是惟一的。这意味着函数名可以重载，即使把函数名相同而参数集合不同的函数看做是不同的函数。函数的返回值不属于函数标识的一部分。

C++函数还可定义为内联函数。编译程序不进行函数调用，而是生成这些函数的目标代码，并插入到客户代码中。当函数不调用时，不会在上下文转换过程中浪费时间。对于关心执行速度的应用程序而言，这是很重要的。

C++只有函数，没有过程。如果应用程序需要使用过程，则可以使用void函数。

如果函数返回一个值，C++允许调用者在调用中忽略这个返回值，而将这个函数作为语句使用。许多C++库函数的返回值很少使用。然而不忽略返回值会好一些。

19.2.4　C++函数的参数传递

在C++中按值传递参数时，参数和局部变量的空间在栈中分配，参数值（变量、表达式或文字）拷贝到为参数所分配的空间中。函数执行时，使用这些数据值；函数终止时，还回栈空间。

在按值传递参数时，数据只能按一个方向移动，即从实际参数到形式参数。改变了的参数值不会传递回来，客户代码作用域中的实际参数不会改变。

对于客户代码空间中的这种副作用，C++支持按指针传递：将指向某个给定类型值的指针作为实际参数传递，而不是将某给定类型的值作为参数传递。从所有角度来看，指针都是变量。它们按值传递，即把指针的值拷贝给形式参数。在执行函数过程中使用指针时，指针包含的内容是客户代码空间中变量的地址。如果有必要，可通过指针来改变这个变量的值。

当函数执行到闭花括号时，撤销指针及其他的形式参数（如果有的话）。这样，函数不会改变指针的值。但这不是问题，因为没有必要改变地址值。按指针传递参数的目的是改变客户代码空间中的变量，这个变量的地址是作为实际参数来传递的。

按指针传递参数比较复杂，程序员必须要协调三个地方的代码：1) 在函数头部和函数原型中使用的指针标记(*)；2) 在函数体内使用的间接引用运算符(*)；3)在函数调用中使用的地址运算符(&)。

为了简化参数传递，C++增加了另外一种参数传递方式来支持调用者空间中的副作用：按引用传递。按引用传递时，代码的不同地方之间进行协调要简单一些：1) 在函数头部使用没有运算符的变量名，2) 在函数体内使用引用运算符(&)，3) 在函数调用中使用没有运算符的变量名。

由于按引用传递这种参数传递方式也支持副作用，它可以代替按指针传递来作为内部数据类型的输出参数（被函数修改的）。

无论是否在函数体内修改其成员的值，C++传递数组的方式都是一样的。同样，代码也需要在三个地方进行协调：1) 函数头中带有空括号的数组名，2) 函数体内的下标标识（或不带括号的数组名），3) 函数调用中不带括号的数组名。

C++的结构对象可以按值传递，也可以按指针或按引用传递。按值传递比较简单：在所有三个地方（函数头、函数体、函数调用中）使用不带修饰符的变量名。但是，它不支持客户代码空间中的副作用。尽管在客户代码空间中不需要副作用，但当传递较大的结构变量时，按值传递也可能会有危害：拷贝这些结构变量会降低程序的速度。

按指针传递参数比较复杂，但它支持客户代码空间中的副作用，而且对于较大的结构也不需要拷贝数据。按引用传递结构参数结合了按值传递参数和按指针传递参数的优点。

对于程序员定义类型的输入输出参数，应使用按引用传递参数的参数传递方式。为了告诉程序维护人员函数修改了哪些参数以及没有改变哪些参数（而不用仔细查看函数细节），设计人员应该在输入参数前使用const修饰符（表示不能被函数改变的参数）。这种强大的方法可以直接通过程序代码而不是注释语句来表达设计人员的意图。

19.2.5　C++中的作用域与存储类别

在C++中采用常规的作用域规则：对象（变量）在由花括号标志的作用域开始处（或中间）定义，在作用域中定义的名字可以在相互独立的作用域中重用。如果在嵌套作用域中重用相

同的名字，内部的作用域对象将隐藏外部作用域中的同名对象。

存储类别（范围）表示运行时为对象分配存储空间的运行时间的跨度，存储类别的名字与它在内存中的地址相关。大多数C++变量属于自动存储类别，这些变量在某个函数或语句块作用域中作为局部变量定义。自动变量的存储空间是当执行到语句块的开花括号“{”时从系统栈中分配的。

自动变量从声明开始起就存在（并可以按名字引用）直到作用域的闭花括号处。自动变量可在定义时初始化。如果它们没有被初始化，就没有缺省的初始值：当在栈中进行分配时，它们的内容是该内存在以前使用时留下的随机数据。

函数的另一次调用（或另一次循环该语句块）可能不使用相同的内存位置。这样，一个自动变量不能在连续调用同一函数之间传递数据。

使用自动变量的好处是在开发小组之间不需要进行协调，就可在不同的函数内重用同一名字。

全局（或外部）变量是在任意函数之外文件开始处（或文件作用域的其他位置）声明的。它们的作用域从声明处开始一直到文件结束。全局变量在程序开始执行之前（main()函数的开花括号处）被分配空间，并在程序终止时（即执行到main()函数的闭花括号处）将空间返回给系统。

全局变量可在定义时初始化，如果一个全局变量没有初始化，则该全局变量缺省为0。

一个全局变量可在任意文件或函数中重新声明为局部变量。在定义该局部变量的语句块作用域中，局部变量名隐藏了全局变量名。局部变量名的内存地址与同名全局变量名的地址不同，不会影响全局变量的内存空间。

使用extern声明可以使全局变量在其他文件中可见。这是在不同文件中实现函数通信的通用方法。

C++的静态存储类别表示了一种折中的设计方案：可在某些上下文中使用这个关键字。

当将全局变量定义为静态变量时，其内存空间在main()函数开始执行之前分配，并在程序终止时撤销，这与一般的全局变量相似。与一般的全局变量不同的是，一个静态全局变量不能在其他文件中作为extern变量：它只在一个文件中的函数可见。从某种意义上而言，这是支持私有数据的一种原始方式。

当局部变量定义为静态变量时，它的内存空间也是在main()函数开始执行之前分配，并在程序终止时撤销的。一个静态局部变量在其作用域中是私有的，它不能被其他的函数访问。但是，当不在其作用域中时，这个局部变量（及其值）仍然存在，再次调用该函数时它又变成可见的。

如果函数初始化了一个局部静态变量，这个初始化工作只能进行一次（即在程序开始处）。当函数终止时，该静态局部变量保留其值，并且在另一次调用该函数时可以使用这个值。这是实现在同一函数的不同次调用之间进行通信的方法。

动态存储类别将分配与回收变量的控制权交给程序员。当程序使用动态内存管理时，程序员要注意避免两个危险：不还回程序拥有的内存，以及使用程序所没有的内存。

不还回程序拥有的内存将导致内存泄漏。内存泄漏会导致删除内存中的程序，尤其当程序不停地执行时。这样程序将崩溃或产生不正确的结果。

使用程序所没有的内存会导致内存讹用，该程序会崩溃或产生不正确的结果，或者在内

存没有发生变化时，产生正确的结果，但一旦内存使用发生变化，程序会突然崩溃（或无任何警告地）产生不正确的结果。

19.3 作为面向对象语言的C++

即使没有面向对象的特点，C++语言也比C语言更为优越。这些优越之处体现在：行尾注释，在作用域中灵活定义变量，变量与指针的符号常量，作用域运算符，类型之间的函数式转换，new与delete运算符，缺省参数，引用参数，函数与运算符重载，内联函数，I/O对象的iostream库以及提高程序质量的（非常多的）相关操作等。

但C++语言对软件工程最主要的贡献是实现了面向对象的特征——类、数据成员和成员函数、构造函数和析构函数、类复合与继承、虚函数、模板与异常等。

19.3.1 C++类

C++类的主要目的是使得程序员可以将相关的数据和操作绑定在一起。这消除了使用独立的全局函数实现基于程序员定义数据类型上的操作所带来的不足。

数据定义为类中的数据成员，操作定义为类中的成员函数。类定义的语法在类作用域内指定了数据成员和成员函数。类作用域是由开花括号和闭花括号来定界的（后有一个分号）。

C++允许类设计者指定程序（客户代码）的其他部分对类成员的访问权限。在定义类对象的任意地方都可以引用公共成员。受保护成员可由类的成员函数或派生类（直接派生或间接派生）的成员函数来访问。私有成员只能由类的成员函数访问。

这样，不属于某个给定类的程序代码只能访问这个类的公共成员。使用友元类和友元函数可以放宽这个规则：它们的访问权限与类的成员函数的访问权限相同。

通常，数据成员定义为私有的，成员函数定义为公共的。这对于维护者而言，将数据和操作绑定在一起；对于客户代码而言，也隐藏了数据实现。有了私有的数据成员和公共的成员函数，客户代码将类看做是函数接口的结合。这样也支持了数据封装和信息隐藏等现代程序设计概念。

具有数据封装性的程序设计可以防止客户代码使用类的私有（或受保护的）数据成员名。这样当数据实现发生改变时，客户代码不会受其影响；同时当成员函数的实现改变时，只要函数接口保持不变，客户代码也不会受其影响。

当然，并不因为使用了C++中的类，面向对象程序设计方法的优越性就自动实现了。如果成员函数只是保存类数据成员的值，并获取这些数据成员以供客户代码使用，在客户代码中使用这样的成员函数将不会使客户代码更加容易读和修改。类设计者必须将任务推向服务器类，而不是上托给客户代码。这种情况可通过增加能够显著实现客户目标的类的成员函数来实现。

以调用服务器类成员函数的形式表达客户代码（而不是以获取和操纵类数据成员的方式），使得客户代码可读性高、便于设计和维护、减少错误等，从多方面提高了程序的质量。

19.3.2 构造函数、析构函数和重载运算符

构造函数和析构函数是特殊的成员函数，它们的函数名不能由程序员随意指定，必须从其所属的类名派生而来。作为回报，客户端代码编程人员不必显式地调用构造函数和析构函

数。反而，在一个类对象的生命期内的某些特定时刻将自动地调用构造函数和析构函数。

当（在栈中或在堆中）创建一个新的类对象之后立即自动地调用该类的构造函数。一个类可以有几个构造函数：这些构造函数的名字都必须相同（即都是类名），但它们的参数列表应互不相同。创建对象时调用哪个构造函数依赖于客户代码指定的参数个数及相应的类型。

在构造函数中，类的设计者指定了如何初始化类的数据成员，以及如何分配其他资源（如堆内存）。

构造函数不应有返回类型，也不能有返回值。如果构造函数要与调用程序进行通信并提供初始化对象出错的相关信息，则构造函数可以抛出一个异常。

比较重要的构造函数有：缺省构造函数、拷贝构造函数及转换构造函数。当创建一个对象而没有提供参数时，调用缺省构造函数。当创建一个对象只给定了一个参数（参数的类型通常是类的某个数据成员的类型）时，调用转换构造函数。

当同一个类的另一个对象用来为目标对象提供初始化数据时，调用拷贝构造函数。当一个类对象按值传递给一个函数时，也调用拷贝构造函数。

当类的对象要动态分配堆内存（或其他资源）时，按值传递对象会产生完整性问题：程序崩溃或错误地运行（或者产生正确的结果直到计算机的内存使用发生了变化为止）。为了防止产生完整性问题，可以用拷贝构造函数来实现值语义。但这将降低程序执行的速度。最好是按引用传递类对象，而不是按值传递。要想使按值传递类对象成为可能，可以把拷贝构造函数声明为私有的（或受保护的）。

（由于作用域规则或delete运算的结果）在撤销一个对象之前，立即自动地调用这个类的析构函数。在析构函数中，定义了如何将对象生命期内所要求的资源还回给系统。

析构函数不能有返回类型和返回值。此外，析构函数也不能有参数。这意味着析构函数不能重载。

使用构造函数和析构函数要求我们对程序的执行看法有一个调整。与其他语言不同，在C++语言中，不仅仅是创建或撤销一个程序员定义类类型的对象，而是在创建对象后经常还有一个构造函数的调用，在撤销对象之前经常还有一个析构函数的调用。

也不能高估构造函数和析构函数对于动态内存管理的作用。与其处理复杂的、相互链接的、由许多各种成员组成的数据结构，程序员宁愿管理简单的、定义良好的、只用处理一两个成员的任务。这样，程序员任务的复杂性降低了，出错的可能性也变小了。

在C++语言中，以同样的方式对待程序员定义类型的对象和内部数据类型的变量。因此，C++语言除了支持函数重载外，还支持运算符重载。这个独一无二的特点允许客户代码对类的对象应用C++运算符。在大多数情况下，这是一种装饰，但在可读性方面是个很好的改善。

有些重载的运算符提供的不仅仅只是表面上的改善。例如，下标运算符可设计来实现下标检查，避免使用C++标准数组时常出现的内存讹用。赋值运算符可设计用来实现值语义，避免内存讹用或内存泄漏问题。如果没有必要使用值语义（一个对象不应该由另一个对象赋值），则应将赋值运算符声明为私有的（或受保护的）。

19.3.3 类复合与继承

当类的对象相互协作时，类提供了最大程度的灵活性。C++支持使用指针和引用实现对象之间的关联：由一个类的数据成员指向另一个类的对象。

在C++语言中，当一个类的对象用作另一个类的数据成员时，也可以使用类复合来实现。C++可以实现的另外一种对象之间的关系是类继承：即一个类作为基类，另一个类作为从基类派生而来的类。

使用类复合，C++把运行时对象的创建过程分阶段进行：首先创建组件对象，然后创建容器对象。由于对象创建时总伴随着构造函数的调用，创建一个复合对象时常变成一系列长长的函数调用，这可能会影响程序的性能。另一个问题是创建一个对象时，可能导致调用一个没有由类实现的构造函数（结果出现语法错误）。

C++提供初始化列表解决这两个问题。它的语法与众不同（再次使用了冒号运算符），但是结果比简单的方法要高级。对那些应该调用的非缺省构造函数而不是缺省构造函数，初始化列表指定了这些构造函数的数据成员的名字和构造函数调用的参数。掌握对象初始化的这种技术并熟练地使用它是很重要的。

使用继承可以将两个类在概念上链接起来。这样，基类中定义的所有成员也成为派生类中定义的成员。此外，派生类可以定义一些其他的数据成员和成员函数，也可以重定义某些从基类中继承而来的成员函数。重定义成员函数时要使用相同的函数名，并对函数体中的内容进行适当的修改。

从客户代码的角度而言，一个派生类的对象结合了在基类中和派生类中定义的功能。使用继承是一种较好的软件设计重用方法，因为不需要编写任何代码，派生类立即可以重用基类中所定义的功能。

因此，一个派生类对象的数据成员既在基类中有定义，也在派生类中有定义；其成员函数在基类中有定义，在派生类中也有定义。客户代码可以访问派生类对象的所有公共成员（数据成员和成员函数），包括从基类中继承而来的成员和在派生类中增加的成员。派生类代码可以访问基类的公共和受保护成员（数据成员和成员函数），但不能访问基类的私有成员。

继承语法以不同的意义重用了关键字public、protected及private（和冒号运算符）。在公共继承模式下，基类中定义的public、protected和private数据成员和成员函数，在派生类对象中依然为public、protected和private。在受保护继承模式下，基类中定义的public成员在派生类对象中变成protected；这样，客户代码就不能访问这些成员了。在private继承模式下，基类中定义的public和protected成员在派生类对象中对象变成private；因此，这些成员不能用于进一步派生。

这是一个复杂的改变访问权限的系统，最好是使用公共继承，保持类之间最自然的关系，即一个派生类的对象也是一个基类对象。这意味着一个派生类对象包含基类的所有数据成员和成员函数。

当实例化一个派生类的对象时，首先创建其基类部分。这涉及到调用基类的构造函数，如果没有缺省的构造函数将出现语法错误。C++语言提供了初始化列表来解决这个问题。其语法与类复合中的初始化列表相似，只是要使用类名而不是组件数据成员名。掌握这个对象初始化方法并好好使用该方法是比较重要的。

当一条消息发送给一个派生类的对象时，编译程序将查找派生类的定义以寻找匹配。如果没有发现匹配的函数，编译程序将搜索基类（或基类的基类等等）。如果找到了匹配，则产生对基类方法的合适函数调用；如果没有找到匹配，则出现语法错误。如果匹配的函数名在派生类中找到了，搜索将终止。如果实际参数与形式参数相匹配，则产生对派生类方法的函

数调用。如果参数不匹配，搜索不会重新开始，而是提示有一个语法错误，即使基类中有完全匹配的参数和函数名的函数。

这意味着派生类方法隐藏了基类中的同名方法，无论这两个方法的标识是什么，因为在不同作用域中不同标识的函数之间不存在函数重载。当实际所调用的函数与客户端代码程序员（或维护人员）认为应该要调用的函数不同时，这常常是微妙错误的根源所在。

由于一个公共继承的派生类对象包含基类对象的所有特性，在任何需要使用基类对象的地方——赋值、参数传递及指针操纵，都可以使用派生类对象。将派生类对象（指针、引用）转换为基类对象（指针、引用）是安全的。这些转换常常不需要显式进行。

将一个基类对象转换为派生类对象是不安全的，只有显式使用赋值运算符或转换构造函数时才能进行这些转换。

将一个基类指针（或引用）转换为派生类指针（或引用）是不安全的。这类转换不能隐式进行，试图进行隐式转换将会标注为语法错误。如果基类指针指向一个派生类对象，将这个基类指针转换为派生类指针是有意义的，但必须使用显式转换才能使编译程序接受这样的转换。如果程序员弄错了（基类指针并未指向一个派生类对象），将会出现运行时错误。

19.3.4 虚函数与抽象类

虚函数扩展了派生类对象中隐藏基类函数的概念。当程序处理属于相似类家族的异构对象集合时，要使用这些函数

每个类中定义了一个函数（使用相同的函数名，如update，和相同的标识）对这些类的对象执行相似的操作（如修改操作），但根据不同类的对象（如SavingsAccount类和CheckingAccount类），处理方式略有不同。

这种设计的目的是想获得多态性的效果，也就是说，将update()消息发送给异构对象集合中的每个对象，或者激活SavingsAccount类的update()消息，或者激活CheckingAccount类的update()消息。要达到这个目标，需从基类（如Account类）中派生出这些相似的类。

这里要遵循几个限制规则：派生模式应是公共的，基类与各个派生类要有同名的函数且同标识的函数（如update）。另外这个方法要定义为虚拟的。这些异构对象集合应作为指向动态分配的派生类对象的基类指针列表来实现。

满足以上限制条件后，发送给基类指针的消息不会激活基类中的相应方法，而是激活由这些指针指向的对象所属派生类的方法。例如，如果基类指针指向一个SavingsAccount类的对象，则调用SavingsAccount类的update()方法。如果基类指针指向一个CheckingAccount类的对象，则调用CheckingAccount类的update()方法。

许多C++程序员都为这个技术而感到兴奋。这毫不出奇，因为该技术使程序很优雅。但是它并不是很重要，因为处理异构集合不是常见的程序设计任务。

通常，这些相似类的基类在应用程序中没有事情可做（如应用程序处理存款账户和支票账户，但不处理没有标识的账户）。将这个方法进一步扩展可使基类变成一个抽象类。

在抽象类中，虚函数被定义为纯虚函数，也就是说，这个函数没有实现，在类定义中的函数原型被“赋值”为0，这是定义纯虚函数使基类成为一个抽象类的语法。应用程序不能创建抽象类的对象。如果程序员错误地创建一个抽象类的对象，将会标注为语法错误。

19.3.5 模板

模板是重用设计的另一种工具。当应用程序需要容器类包含不同类型的组件时，最好是避免每个组件类型的重复设计——因为这些类几乎是相同的。

C++的模板允许程序员创建这样的一个类，组件类型定义为这个类的一个参数。在实例化这个类型的对象时，客户代码提供实际类型的名字，编译程序将生成类目标代码，其中类参数的每个实例由客户代码所给定的实际类型名代替。

模板语法相当复杂，要在尖括号中使用参数列表。实例化对象的语法也相当复杂，客户代码也要在尖括号中给定实际类型。成员函数的实现也要用尖括号括起形式参数列表和作用域运算符中的类名。

尚不清楚各个应用程序从使用模板中获得多少益处。模板这个方法相当新，并不是所有的编译程序都支持使用模板，模板的使用经验也还不够。但是，C++标准模板库使用模板实现了诸如表、队列、向量及哈希表等类似的数据结构。这些类都设计良好并已优化过，在应用程序中使用它们肯定会受益非浅。因此，理解实例化和使用模板对象的语法比较重要。

19.3.6 异常

异常也是C++语言中的一个新特点，用来将主要算法的处理与异常和偶然情况的处理分隔开来。一般来说，它将难懂的代码清晰化，有助于简化主要的处理算法，可以降低整个程序的复杂度。

当抛出异常时，创建一个内部数据类型的值或程序员定义类型的对象，并将这个对象从发现异常情况的地方发送到处理异常的地方。这个数据值（或对象）携带着用于异常处理的有关信息，例如表示一条错误消息的字符串。

使用异常时，要编写声明异常、抛出异常和捕获异常的语句。

在声明一个异常时，要在函数头和函数体之间使用关键字throw，在throw之后是用括号括起来的这个函数所能抛出的异常类型列表（多个异常之间用逗号分开）。如果定义了函数的原型，则在该函数的参数列表的闭花括号与最后的分号之间，插入throw关键字以及抛出的异常类型列表。

当抛出一个异常时，程序员使用同样的关键字throw及某个参数（这个参数可能是内部数据类型的某个数据值，也可能是程序员定义类型的某个对象）。通常，在某个条件语句中使用抛出异常的语句：对某些条件进行判断，当条件满足后返回真，抛出异常，并通知异常处理程序发生的情况。

如果throw语句抛出的是一个程序员定义类型的异常，则相关数据应随着对象进行传递，异常类应该定义一个构造函数，以接收必要的数据并初始化对象的数据成员。如果对象不需要任何数据，则调用缺省的构造函数；与其他用缺省构造函数创建对象的情况不同，这里必须要在对象名后使用空括号。

当捕获一个异常时，要使用比较复杂的语法结构——带有能抛出异常语句的try语句块，try语句块后是一组catch语句块。每个catch语句块被设计来只处理某个类型的异常。

try语句块只是一个由try关键字开头的未命名的语句块。一个catch语句块是由catch关键字及只有一个参数的参数列表开头的语句块。catch参数的类型是这个catch语句块被设计来处理的异常类型（内部数据类型或程序员定义类型）。

如果try语句块中的语句没有抛出任何异常，那么try语句块将终止，catch语句块也会被忽略，并执行下一条语句（如果有的话）。如果try语句块中的某条语句抛出了一个异常，则将忽略从这条语句到try语句块结束之间的所有语句，即使捕获了这个异常而且进行了处理后也不会执行这些语句。

然后，依次检查catch语句块，在参数列表中查找与异常抛出的值类型相匹配的参数类型。如果一个catch语句块中没有匹配的类型，则查找下一个catch语句块。如果某个catch语句块的参数类型与抛出的异常类型相匹配，则执行这个catch语句块，对异常进行处理。随后，其他的catch语句块将被忽略，然后执行最后一个catch语句块后面的那条语句。

如果没有找到匹配，将忽略try-catch序列后面的语句，包含这个try-catch序列的函数也将终止执行。在终止之前，抛出没有捕获的异常，然后继续查找该函数的调用程序。如果找到了相应的catch语句块，程序将正常执行。如果没有找到，则继续查找，一直找到main()函数，程序将终止执行。

由于可以将try-catch序列放在任意函数中，也可以按任意方式结合使用，因而异常处理机制将导致代码模式比较复杂，经常很难看出异常在哪里处理，程序如何继续执行。因此，并不十分清楚使用异常是否的确可以导致主流处理的简化和异常处理的容易理解。

使用异常处理机制的优点是将发现错误的代码与处理错误的代码分别放在不同的地方，所有必要的信息可以放在throw语句抛出的异常对象中，从发现错误的地方传送到处理错误的地方。另一个优点是当函数中没有合适的catch语句块而终止执行时，在栈中要删除的所有对象将调用析构函数。这意味着如果程序从错误中恢复过来后继续执行，不会出现内存泄漏问题。

19.4 C++及其竞争对手

从某种意义上而言，C++语言与目前正使用的任何一种语言都存在着竞争，包括FORTRAN、COBOL语言和PL/I、Ada语言。

19.4.1 C++与传统的语言

当然，即使是历史比较悠久的传统语言，也有它们各自的长处。例如，FORTRAN语言的优越性体现在科学计算上，它所提供的库有助于进行科学计算的程序员以更简单的方式进行处理。

就灵活的输出格式而言，COBOL和PL/I要比C++语言优越：C++程序员必须花费大量的精力才能获得类似的结果，尤其是使用iostream库时。而如果使用传统的标准库，格式代码可以更为简明。然而，仍容易出错，也比较复杂。

Ada语言的特点是用于并发程序设计，并为实现简单对象提供了包，利用这些特点可以实现基本的面向对象设计。

然而，这些语言都不支持复合、继承以及虚函数。对于编写大型应用程序而言，C++语言的这些面向对象的特点使得C++语言比其他语言要更优越。

19.4.2 C++与Visual Basic

C++语言的另一个越来越重要的有趣的竞争对手是Visual Basic。多年来，Visual Basic已

从一种非常简单的语言演化为一种功能强大而灵活的语言。

Visual Basic支持的输出格式功能可以与COBOL和PL/I的功能相媲美，比C++语言的功能要好得多。Visual Basic为程序员使用图形用户接口（Graphical User Interface, GUI）建立交互式的输入输出提供了一种快速简捷的方式。而对于C++语言来说，程序员要取得类似的效果，必须学会窗口库，而目前市场上的所有库都非常难以学习和使用。

Visual Basic也支持某些面向对象的特征，尽管在这些方面它不如C++语言，但Visual Basic比C++语言更容易学习。学习用Visual Basic开发有意义的应用程序比学习用C++开发要快得多。

但是，C++语言将面向对象的特征集成到语言中，这方面比Visual Basic要好得多，Visual Basic只是将它作为附带特征。。

19.4.3 C++与C

C++语言的另一个竞争对手是它自己的祖先——C语言。

C语言仍然是开发实时和嵌入式系统的程序员的一种选择。这些程序员担心C++语言中像虚函数、模板、iostream库及异常这样复杂的概念，他们认为所有这些特征都会影响程序性能，并增加可执行代码的大小。对于实时和嵌入式系统而言，程序性能和可执行代码的大小是主要的问题。因此，这种类型的应用程序很少使用C++语言开发。

我们并不这样看。C语言比较简捷，有助于以简明、易于表现的方式编写程序代码。但这些代码方式有时会让读者相当迷惑，尤其当程序员试图优化程序的性能而使目标代码最小化时。

结果，使用C语言开发实时和嵌入式系统的组织会面对复杂的内存使用方案，以及用于动态内存管理的复杂的全局数据结构和纠缠不清的调用规则等。这些系统的设计太复杂，因此在紧张的发布日程压力下不可能合适地文档化系统。训练新的人员要花很长时间，给已有员工的生产力水平带来负面影响。对系统其他部分的不正确的开发决策和不完整的理解将导致设计错误和维护错误，要纠正这些错误代价很高。

我们并不肯定这种现象是否是普遍存在的，但是至少在好几个正在努力从C向C++转换的公司中看到这些现象。

显然，使用C++语言中的类将私有数据成员和公共成员函数绑定在一起，有利于模块化程序设计，同时也不会影响程序的性能和目标代码的大小。正确地使用构造函数和析构函数可以消除内存管理中的错误，而且也便于查找错误。同时，适当地使用虚函数也不会对程序的性能和目标代码的大小造成大的影响。

毕竟，多态性可以在任何语言中实现。当在C语言中实现多态时，要分配额外的内存，以存储程序正在处理对象的类型信息，而且要花一些时间去理解对象的处理方式，但C++语言为程序员将这些复杂性封装起来，不会浪费额外的资源。

不幸的是，无论程序员是否使用了模板、iostream库、异常或运行时标识（run-time identification）等，许多C++编译程序都会生成很多的目标代码。对于这些C++语言中高级特征的使用限制，要求与编译程序的供应商协商，以便使程序员在每个创建过程中可以选择需要包含哪些特征。我们认为，最终所有使用C语言的程序员都会转向使用C++语言，即使不是每个应用程序都会用到C++语言的所有特征。

19.4.4 C++与Java

目前，Java可能是C++语言的最大竞争对手。Java与C++语言比较类似，都是从C语言演化而来，Java的大多数面向对象语法都借用了C++语言的语法。

与C++语言类似，Java也支持带有数据成员和成员函数的类，支持构造函数、虚函数及异常等。Java程序代码是由相互提供服务的协作对象组成的集合。Java还支持类复合、类继承及多态性。Java鼓励重用程序员定义的类和图形可视化用户接口的类库。

与C++语言不同，Java并不与C相兼容，这样Java避免了C++语言许多从C语言继承而来的容易出错的特征。此外，Java也不包含C++语言中导致目标代码变大及与软件工程方法相违背的特征。

有些人认为Java没有指针，因此Java源代码比C++源代码要简单得多。但事实并非如此，Java中有指针，例如在运行一个简单程序时，在程序放弃运行之前，在屏幕上可以看到“空指针异常”消息。

Java有一个与C++语言的显式的new运算符，但没有显式的delete运算符，而是使用垃圾回收。这打破了C/C++语言的思维方式：在内存管理上所花的时间越少，留给其他算法的时间就越多。

使用垃圾回收和在运行时使用解释器，使得Java比C++语言要慢得多。几年以前，这也许意味着一门语言的终结。但如今并非如此，性能固然重要，但不再是评价一门程序设计语言的最重要标准，可移植性、健壮性及简单性要重要得多（假设语言支持现代面向对象软件工程方法）。

Java具有可移植性：所有Java数据类型的大小在所有机器上都是标准大小。一个整数总是4个字节，它也不一定是给定平台上最快的类型，但是在所有的机器上程序以相同的方式运行。不可以在signed和unsigned数据类型之间进行选择：数值类型总是有符号的，布尔类型和字符类型总是无符号的。Java中的标识符可以是任意长。

Java具有健壮性：不允许在数值类型之间进行隐式转换；只允许在数值类型之间使用显式转换，而不能在数值类型和布尔类型之间进行显式转换。Java与C/C++语言不同，关系运算符和逻辑运算符返回布尔类型的“true”或“false”，而不是返回整数1或0。这样，将比较运算符==写成赋值运算符=时会标记为语法错误，这是程序设计中容易忽视的一个主要错误源。

Java也比较简单：它没有C++语言中一些容易出错的特征：如重载运算符、类属模板、多重继承、按指针传递参数、与指针相关的没有保护的数组操作、友元类和友元函数、全局函数及全局变量（Java中的每个函数，包括main()都必须是某个类的成员）。

Java没有宏、包含文件及条件编译预处理器，也不必用函数原型——这是C++语言中使人头疼的重要根源。Java编译程序知道库所在的位置，程序员不必指定位置。

Java还实现了C++语言中所没有的面向对象特征：所有函数缺省为虚函数，不仅可以在派生类中隐藏基类函数，当函数标识不同时，也可以在派生类中重载基类函数。还可以将几个类集成为一个包。当需要使用一个类的对象时，可以使用另一个类的对象来代替，Java中提供了界面结构（interface construct）对此进行控制，这比继承控制要好。Java还支持线程的概念，有助于实现同步。

Java编译程序产生所谓的要求运行解释器的字节代码（bytecode），而不是依赖于机器的

目标代码。这看起来像是一种负担，其实不是，这意味着Java的字节代码可以运行在任意有Java解释器的平台上：在Solaris机器上编译的程序可运行在任意的UNIX、PC或Mac机器上，而不需要任何修改。C++程序员无法奢想实现这种可移植性。

因此，Java作为一种因特网语言是成功的：在某个平台上生成的字节代码可以从另一个平台上下载并运行，甚至不会询问是什么服务器平台。在异构网络中使用Java与在同构网络中一样简单。

Java提供了一个庞大的GUI类库。学习这些类的确不是件容易的事，但是初学者几乎可以像编写Visual Basic应用程序那样容易地编写出有意义的Java应用程序，比使用C++编写应用程序要简单快捷得多。

在因特网上，Java显然是个赢家。对于终端（backend）处理，C++则是个赢家，大多数情况下都是因为它的性能优越。但如今的形势不像几年以前那样明显了。

19.5 小结

迄今为止，我们已看到了C++语言的所有特征，无论是它的优点还是它的缺点，我们都真实地进行了阐述。无论是好的、不好的还是危险的特点，我们都已如实讨论。

但首先最为重要的是，有效地使用C++语言需要改变程序设计的思路。我们应该考虑不同程序部分之间的任务划分，并将这些任务推向服务器类。我们也应该注意要通过程序代码而不是注释语句来将设计的意图传达给维护人员。我们还应该思索在需要使用一个类的对象地方使用另一个类的对象。

运用这些软件工程的基本原则，我们可以创建出健壮的、可移植的、可重用的、可维护的C++应用程序。而且从中也会获得无穷的乐趣，因为C++语言是一个有趣的语言。

索　引

(按字母数字排序)

#define指示符，24~26
#else指示符，25~27
#endif指示符，25~27
#ifdef指示符，25~27
#ifndef指令，27
#include指示符，24~25，26
　程序员定义类型，155
0终止符，138
2000年问题，4

A

access()方法，派生类，600~603，604~606
Account类，487~506，651~659
　applyfee()，488，490
　createAccount()，653
　deposit()，486，495~496
　getBal()，495~496，505
　kind枚举类型，488~489
　main()函数，654
　payInterest()，488，490，501，505
　printList()函数，655
　processRequest()，653，654~655
　withdraw()，486，495~496
　存款账户，490~491，495~502，506~508
　继承层次，例子，497~498，523~524，528~529
　交易账户，499~501
　客户代码正确性的运行时测试，例子，490~491
　提高的继承层次，例子，530~531
　用程序员命名的方法处理，例子，653~654
　用重载运算符方法来处理，655~656
　在类中合并不同功能，例子，487~488
　支票账户，490~491，495~502，506~508
Ada，9~10，14
addComplex()，349~358
Address类，537~539
allocate()，423
allocateAccounts()，254
APL（一种编程语言），9，405
appendCust()，572
appendItem()，572
ASCII码，57
atof()库函数，147
atoi()，147
atol()，147
avg()，248

B

bad_alloc类，756~757
Booch, Grady，14
break语句，20，117~120

C

C++，9~10
　抽象类，781~782
　聚集类型，773~774
　　结构，774
　　联合，774
　　枚举，774
　　数组，773~774
C++标识符，用于数组元素，131
C++标准库，401
　异常，756~757
C++标准模板库（STL），587~588
C++词法作用域，167
C++代码元素，组合，484
C++函数，41~47
　参数传递，230~255，774~775
　　按引用调用，238~241
　　按值调用，230~231，396~400
　　按指针调用，232~238
　　结构，241~246
　　数组，246~249
　参数的提升和转换，230~231
　从……返回值，251~255
　存储类别，776~778
　定义，227，651
　过程，228

函数调用，228~230
函数定义，228
函数名重载，261~266
函数声明，227~228
类型转换，249~250
内联函数，255~257
设计，227
有缺省值的参数，257~261
作为模块化工具，226~230，774~775
作用域，776~778
C++函数名称，作用域，168
C++继承
用来链接相关类，495~497
语法，497~503
基类的不同派生模式，498~501
基类和派生类的对象，501~503
C++类，278
const关键字，340~344
不同存储方式的类对象，定义，313~314
对象实例初始化，319~335
构造函数/析构函数的调用时间，330
拷贝构造函数，323~325
类作用域，331~333
缺省构造函数，321~323
析构函数，326~330
用运算符和函数调用的内存管理，333~335
转换构造函数，325~326
作为成员函数的构造函数，319~321
基本的类语法，304~319
静态类成员，345~351
static关键字，347~348
初始化静态数据成员，348~349
静态成员函数，348~349
特征，345~347
作为类使用全局变量，340~348
开花括号/闭花括号，307~308
类成员，控制对……的访问，314~319
类构造的目标，304
绑定数据与操作，304~307
消除名字冲突，307~310
在类之外实现成员函数，310~313
主要目的，778
作为数据和函数集合，129
作用域，307~308
C++内部数据类型，769~771
C++全局函数，775
C++数组，定义，131~133
C++异常，使用的优点，757
C++异常的语法，736~748
捕获异常，736，738~743
错误相关的源代码，组织，736~737
抛出异常，736，737~738
声明异常，737，743~745
异常处理机制，736
异常处理机制，相关操作，736
重新抛出异常，745~748
C++语法，定义变量，129
C++指针
初始化，189
作为类型的变量，187~191
C++转移语句，117~127
break语句，117~120
continue语句，120
exit语句，121~125
goto语句，120~121
return语句，12.~125
switch语句，125~127
catch关键字，738
catch语句，736
char关键字，31~33
checkIn()，572，581
支票账户，490~491，495~502，506~508
checkOut()，572，583
checkParen()，304~305
Circle类，308
getArea()，542，543，544~545，550~553
getLength()，542，543，544~545，550~553
getRadius()，542，550~553
set()，542，550~553
class关键字，318，319，519~520
Client()，334~335
closestPointPtr()，338~339，341~342
closestPointRef()，338，339，341
closestPointVal()，338~339，341~342
COBOL，783
C++，783
compareTime()，55
const修饰符，241~246，246~249
const_cast运算符，342
const关键字，87，340~345，348，368~370，396，458，459
const关键字，444~445
continue语句，20，120
Convey, John，5

copyAccounts(), 250
CopyData(), 326
createNode(), 259
Cube类, 311
custIdx, 572
Customer类, 565~566
 Cylinder类, 305~306, 307~308, 309~310, 311~313
 类的实现, 568
 类规则说明, 567
C语言, 9, 613
 C++, 783
 目标, 18~21
 与C++相比, 17~21

D

Database类, 565
DebugPrint(), 704
dec, 189
delete[], 651
delete运算符, 191~193
deposit(), 495~496, 500, 505
Development Studio (Microsoft), 24
DictEntry类, 712, 712~713, 726~727
displayMilitaryTime(), 48~51, 55~56
displayResults(), 55~57
displayTime(), 48~51, 55~56
double关键字, 31~32
do-while循环, 104, 111~114, 772
 没有预读, 112
 实现字符输入, 113
 在条件语句中带有赋值运算, 113
do循环, 20
draw(), 497, 612~614, 615
dynamic_cast运算符, 764~766

E

EBCDIC编码, 57
Edt, 23
Eiffel, 186
else关键字, 85, 89, 90
Emacs, 23
enum, 775
enum关键字, 157, 161
errno, 730
exit语句, 121~125
explicit关键字, 593
extern关键字, 33~34

F

fail(), 574
File类
 类规则说明, 574
 实现, 574~575
FileOutItem类, 584
findCustomer(), 578, 581
float, 230
 浮点类型, 64~66, 769~771
float关键字, 55
FORTRAN, 783
 和C++, 783
for循环, 20, 114~117
 实现事务处理, 114
 在初始化表达式中使用逗号运算符, 115
fraction(), 731~736, 741, 744~745, 746~747, 751, 753~756
free(), 17
 内存管理, 193~194

G

get(), 142, 203~204, 537~538, 684
getArea(), 542, 543, 544~545, 550~553
getBal(), 495~496, 505
getBalance(), 167, 168, 169~170, 178
getCustomer(), 572, 574~575, 578
getDistPtr(), 337~338
getDistRef(), 337~338
getHeight(), 315~316
getId(), 582
getInt(), 675, 678~680
getItem(), 572, 574~575, 578
getKind(), 622
getLast(), 299
getLength(), 542, 543, 544~545, 550~553
getLine(), 220, 258, 575
getOther(), 589, 593
getPtr(), 336
getQuant(), 582~583
getRadius(), 315, 542~543
getRef(), 336~337

getSize(), 660~661
getVolume(), 305~306
getX(), 344
GNU编译程序, 51
goto语句, 20, 120~121

H~I

hex, 189
History类, 469~482
if-else语句, 772
ifstream, 217, 218
if关键字, 84
if语句, 20, 694
ignore(), 142
initialize(), 512, 514
initStore(), 300
inline修饰符, 256
inStock(), 583
int关键字, 31~32
Inventory类, 566, 570
 类规则说明, 570
 类实现, 571~572
inverse(), 731~736, 741, 744~745, 746~747, 751, 753~756
iomanip头文件, 198
iostream, 36
isEmpty(), 299~301
isLater(), 55
isLeft(), 299
isRight(), 299
isSamePoint(), 445~446
istream类, 688~689, 692
itemIdx, 572
Item类, 565~566, 582
 类的实现, 568
 类规则说明, 567

J~N

Jacobson, Ivar, 559~560
Java, 9~10, 783~786
 和C++, 783~786
Lisp, 186
loadData(), 577, 584
longjmp, 730~731
Long修饰符, 60~62
Matrix类, 683~687
 get(), 684
 make(), 683~684
 printMatrix(), 684
 使用Matrix类作为方阵的容器, 684~685
 重载函数调用运算符, 686~687
Microsoft Visual C++编译器, 24
modify(), 404, 405
modifyString, 662~663, 666~667
move(), 443, 504
Negative Denom类, 750~756
new运算符, 191~193
normalize(), 371~372, 374

O

ofstream, 216, 218
OMT 见对象建模技术
OOSE 见面向对象软件工程
operator+(), 356~358, 397, 398~399
operator+关键字, 364, 366, 373~374
operator关键字, 651~652
ostream类, 691
Other类, 589, 594

P~Q

P.Chen, 559
Pascal, 228, 613
PayInterest(), 488, 490, 501, 505
peek(), 204
Person类, 616~622
PL/I, 19
 和C++, 783
Point类, 345~346, 348~349
pop(), 512~513
precision(), 197~198
print Accounts(), 172~173, 176~177, 179, 241~246, 244~245
printAverage(), 179~182
printCaption(), 180, 181, 182, 183
printCylinder(), 305~306
printItem(), 584
printMatrix(), 684
printRental(), 582
printString, 660, 662, 663
Private关键字, 316, 495~496, 498~499, 530~532, 537

processItem(), 578
processTransaction, 12
protected关键字, 316, 495~496, 498~499, 530~532, 537
public关键字, 316~318, 798~799
push(), 512~513
quantity(), 346, 348~349, 351

R

Rational Rose, 559~560
Rational类, 396~399
 测试用例, 374~375
Rational软件公司, 559
read(), 617, 618~619
Rectangle类, 350, 377, 464~466
 const关键字, 444~445
 pointIn(), 445
 构造函数, 443, 444~445
 客户端代码的设计, 443~445
regEvent(), 260, 266
registerEvent(), 260~261, 266
reinterpret_cast运算符, 761
reset(), 660~664
retrieve(), 514
return语句, 55~57
Rumbaugh, James, 559~560

S

samePoints(), 351
saveCustomer(), 575, 578, 584
saveData(), 578
saveItem(), 575, 584
saveSymbol(), 299
SavingsAccount, 490~491, 495~502, 506~508
scaleCylinder(), 305~306
set(), 504, 513, 660, 711
setAccount(), 311
setCylinder(), 305~306, 307~308, 309~310, 311~313
setf()function, 197~198
setHeight(), 315~316
setInt(), 675, 678~680
setjmp, 730~731
setOther(), 590
setPoint(), 342~343
setRadius(), 315
setTime(), 48~51, 55~56
setw(), 40, 198
short修饰符, 60~62
show(), 397
 基类, 600~603, 604~606
show_name, 327
showComplex(), 363~364, 366
signed修饰符, 59, 62
sizeof运算符, 61, 67, 191, 659
Stack类, 722~723
static_cast运算符, 757~761
static关键字, 347~348
stdio.h库, 37
StoreClerk类, 565
Store类, 565, 572, 575, 576, 577
 Main () 函数的实现, 578~579
 类规则说明, 575
 实现, 576~577
strcat(), 140, 143, 203, 204, 229
strcmp(), 140~141, 145~146
strcpy(), 140~141, 144, 203, 204, 229, 329
String类, 395~438
 带有重载运算符, 430~432
 动态分配的堆内存, 402
strlen(), 141, 203~204, 229
strncat(), 144
strncopy(), 144
strncpy(), 404~405
strtod(), 147
strtol(), 147
struct关键字, 151~152, 313~316
swapAccounts(), 241~246
swapTransations(), 249
swich语句, 20, 125~127
symbolsMatch(), 299

T

template关键字, 704, 706
this关键字, 339
throw关键字, 67, 737
 声明异常, 743
TimeOfDay类, 48~51, 55~56
trim(), 575
try关键字, 738
typedef功能, 150

typedef关键字，150~151，698，700
typeid运算符，766~767

U

UML 见统一建模语言
union关键字，157
和标志域一起使用加强访问的完整性，159~160
unsigned修饰符，59~60，62
using namespace指令，25

V~Z

Vi，23
virtual关键字，629
VisiblePoint()，509~510，514，532，533
Visual Basic和C++，783
void关键字，44
while循环，20，105~111
使用预读输入字符，110
无限次循环，106
循环条件中带有赋值运算，110
用一个负数或0作为标记来实现，107
用预读来实现，109
width()库函数，147~148，198
withdraw()，495~496，500，505
write()，624~627
ZeroDenom类（例），749~750，752~756

（按拼音排序）

A~B

按位逻辑运算符，70~72
八进制数，57
包，9
保证，2
编程，1
遍历，104~117
do-while循环，104，111~114
没有预读，112
实现字符输入，113
在条件语句中带有赋值运算，113
for循环，104
while循环，104，105~111
使用预读输入字符，110
无限次循环，106
循环条件中带有赋值运算，110
用一个负数或0作为标记来实现，107
用预读来实现，109
标号名，168
标量，129
标量变量，131~132
标量类型，32
标志域，联合，159~160
标准，2~3
表达式，34~40
按位逻辑运算符，70~72
逗号运算符，77
赋值运算符，75~76
高优先级运算符，66
关系运算符，72~73
混合型，77~82
逻辑运算符，73~75
算术运算符，67~69
条件运算符，76~77
相等运算符，72~73
移位运算符，69~70
运算符总结，66
捕获异常，736，738~743
例子，740
布尔关键字，55
布尔类型，值，770
布尔值，64

C

参数，44
参数传递，230~255，774~775
按引用传递，238~241，774
按值传递，230~231，396~400，774~775
按指针传递，232~238，774~775
基本规则，241~242
简单变量的总结，241
参数匹配，739
操纵符，189
操作数，35
操作系统，2
常量数据成员，458~459
成员的初始化列表，453
程序复杂性，组件，10~11
程序函数，612
程序开发工具，51~54
程序开发人员之间的交流障碍，6
程序设计语言特征，17~21

程序员，摆脱，5~7
程序员定义的拷贝构造函数，420~423
程序员定义类型的结构定义，151~152
程序员定义数据类型，129
 聚集，129~165
程序作用域，167
重新抛出异常，745~748
 在catch语句块中，例子，747~748
重载运标符<<，691~693
重载运算符>>，778~779
重载运算符
 定义，651
 一元，652
 运算符优先级，652
 作为类成员，364~369
抽象类，781~782
出租商店的实例分析，565~580
 类及其关联，565~580
初始化
 C++指针，189
 变量，152~153
 对象，15~16
 对象实例，319~335
 多维数组，136
 复合对象，446~457
 结构变量，152~153
 静态数组成员，348
 数组，129，130、131
初始化列表，453
处理数组，使用指针，193
传统的语言，783
 重载运算符，778~779
纯虚函数，636~639
词法作用域，167~175
磁盘文件的输入和输出，215~223
 从文件输入，217~221
 输出到文件，215~217
 输入/输出文件对象，221~223
存储类别，175~186，330~331，776
 自动变量，176~178，776
 定义，174
 动态存储类别，777
 静态变量，182~186
 外部（全局）变量，178~182，774~775
错误相关的源代码
 组织，736~737
 在异常处理方法中，736

D

代码，4
代码可理解性和独立性，269
代码重用
 借助服务重用，547~550
 例子，548~549
 客户-服务器类关系的示例，542~545
 通过公共继承并重新定义方法的重用的例子，555~556
 通过继承，550~554
 例子，550~551
 通过受保护继承，例子，553~554
 选择一种技术，542~557
 以重新定义函数的方式继承，554~556
 运用智力，545~547
 例子，546
单独的非成员函数，作为模板，728
单元测试阶段，瀑布方法，7
点选择运算符，774
 定义，30~33
动态绑定，633~636
 传统方法，615~622
 规则总结，635
 面向对象的方法，622~629
 虚函数，629~633
动态变量，166
动态存储类别，777
动态结构，207~215
动态内存管理，142~143，660
 类使用继承，538
 链式结构，208
动态数组，197~207
 容纳无限的输入串，202~203
动态数组常见局限，208
逗号运算符，77，772
堆，166，186~215
 分配内存，191~194
 将数据读入到分配在堆的数组中，199
堆结点
 使用链表，213
 为了从磁盘文件中读取数据，219
堆内存的动态管理，402~406
对象，16~17
 定义，16
 复合，353
 数据，16~17

优点，17~18
支持，21
对象初始化，15~16
对象建模方法（OMT），559
对象库，559
对象实例初始化，319~335
构造函数/析构函数的调用时间，330
拷贝构造函数，323~325
缺省构造函数，321~323
析构函数，326~330
转换构造函数，325~326
作为成员函数的构造函数，319~321
多继承，640~649
定义，642
二义性，645~647
访问规则，643~644
构造函数/析构函数，644~645
类之间的转换，643~644
用处，648~649
有向图，647~648
多态性，20，497，517，614
多维数组，136~138
耽用内存，135~136

E

二进制数，57
二维数组
操作，137
引用一个元素，138
二维字符数组，144~146
搜索，145
二元运算符，772
二元运算符，651，659~674
后缀重载运算符，666~668
减量运算符，659~666
增量运算符，659~666
转换运算符，668~674

F

返回对象
在客户端代码中使用，336~340
返回对象，338~340
返回指针和引用，336~338
方法　见C++函数
访问规则，多继承，643~644
非数值类
重载运算符，400~414
String类，401~402
保护程序完整性，409~413
保护堆数据对象，406~407
堆内存的动态管理，402~406
防止内存泄漏，408
重载的串接运算符，406~407
非相关类之间的转换，589~594
强类型，591
弱类型，591
指针或引用之间的转换，593~594
转换构造函数，592~593
转换运算符，594
封装，14~15
缺点，299~301
主要优点，14
服务器对象与客户对象之间的联系，13
浮点类型，64~66
浮点数类型，32
父类，495
复合
优缺点，556~557
在继承和，541~586
复合对象，353
没有调用多余的构造函数，454~455
复合类（对象），439~482
初始化，446~457
使用成员的初始化列表，453~457
使用组件的缺省构造函数，448~453
定义，441
访问数据成员
方法参数，445~446
类数据成员，443~445
具有特殊属性的数据成员，457~465
常量数据成员，458~459
引用数据成员，459~461
用对象作为其类自身的数据成员，461~463
用静态数据成员作为其类自身的数据成员，463~465
类聚集，441
C++语法，442~443
容器类，465~482
嵌套类，478~480
友元类，480~482
用类对象作为数据成员，440~446
复数类，354~358，360~367，374~377

对象执行的操作例子，355
复习，22~54
复杂语句，36~38
嵌套，772
赋值，37
赋值运算符，75~76

G

岗哨值，131
高级程序员，5~7
高内聚的函数，命名，270
高优先级运算符，66
公共继承，507~511
访问一个派生类对象的基类成员，509~510
公共派生，508~509
构造函数，395~438，778~779
调用时间，330
对派生类而言，532~545
初始化列表，535~537
多继承，644~645
赋值运算符的重载，427~435
链表达式，430~433
内存泄漏，428~429
系统提供的赋值运算符，427~428
性能的考虑，433~434
自我赋值，429~430
拷贝，323~325
按值返回，423~427
程序员定义，420~423
拷贝语义和值语义，419~420
完整性问题的补救措施，415~419
有效的局限性，427
类型转换，325~326
缺省，321~323
语法，326
作为成员函数，319~321
固定大小的数组，常见局限，208
关键字return，44
关系运算符，72~73，771
规则说明，模板，724~728

H

孩子类，495
函数，41~47
void，229
定义，227
函数体，41~43，44~45
函数头，41~43，44~45
函数作用域，167，168，172
将数据与函数绑定在一起，12~14
模板，728~729
使用的优点，41
友元，383~394
运算符，353~394
函数参数，702
函数调用，41~47
函数调用运算符，683~687
函数定义，228，6.2
函数名，612
函数名，作用域，168
函数名重载，261~266
函数声明，227~228，612
后缀加1运算符，例子，667~668
后缀运算符，68~69，771
后缀重载运算符，666~668
混合参数类型，377~383
混合型，77~82
混合型表达式，77~82

J

基本程序结构，22~24
基础编程语言，9
基类，495，508，516~517
show()方法，600~603，604~606
传递指针与引用参数，609~610
使用指针访问对象，603~604
基类指针（引用），转换成派生类的指针（引用），781
集成测试阶段，瀑布方法，7
集成开发环境（IDE），23
继承，439，554~556
调整对Derived类中的成员的访问权限，517~518
对基类和派生类服务的访问，503~506
对派生类对象的基类成员的访问，506~520
多个，640~649
定义，642
二义性，645~647
访问规则，643~644
构造/析构函数，644~645
类之间转换，643~644
用处，648~649

有向图，647~648
非相关类之间的转换，589~594
强类型，591
弱类型，591
指针或引用之间的转换，593~594
转换构造函数，592~593
转换运算符，594
公共继承，507~511
名字重载与名字隐藏，522~525
派生类所隐藏的基类方法的调用，525~529
缺省继承方式，518~520
使用进程改进程序，529~532
使用异常，752~756
受保护继承，511~515
私有继承，515~517
通过继承的代码重用，550~554
例子，550~551
通过继承相关的类之间的转换，594~612
安全/不安全的转换，595~599
对象的指针与引用的转换，599~607
指针与引用参数的转换，607~612
用来链接相关类，495~497
优缺点，556~557
有预定义的函数，554~556
语法，497~503
在继承和复合中选择，541~586
作用域规则和名字解析，520~522
加1运算符，68~69，659~666，771
对指向内部数据的指针使用运算符，661~662
作为消息发送给对象的例子，664~665
间接超类，496
间接引用运算符，187
间接引用指针，195，208，228~229
间接运算符，187
减1运算符，68~69，659~666，771
简单变量，129
简单的数据类型，32
将数据与函数绑定在一起，12~14
将语句合并在语句块中，84
交易账户，499~501
结构，21
变量：
操作，154~155
创建/初始化，152~153
程序员定义类型的结构定义，151~152
传递，241~246
动态的，207~215
结构的域，153
结构定义，774
元素，153
在多个程序中定义，155~157
组件的层次结构，153~154
作为异构聚集，151~152，157，161，318~319，616
作为引用参数传递，244
结构变量，774
借用服务
代码重用，547~550
例子，548~549
警告信息，编译程序，54
竞争，783~786
静态，作为其类自身的成员，463~465
静态绑定，613~615
规则总结，635
静态变量，182~186
局部，185
全局变量，184~186
静态成员，模板，722~723
静态类成员，345~351
static关键字，347~348
函数，348~349
静态数据成员的初始化，348~349
作为类特征使用全局变量，345~347
局部指针，未初始化，189
聚集
程序员定义数据类型，129~165
类，440，441
C++语法，442~443

K

开发环境，51
拷贝构造函数，414~427
按值返回，423~427
程序员定义，420~423
拷贝语义和值语义，419~420
使用拷贝函数从函数中返回对象，424~425
完整性问题的补救措施，415~419
用另一个对象的数据去初始化一个对象，421~422
有效的局限性，427
拷贝构造函数，323~325
拷贝实例化，467
拷贝语义，419~420

可维护性，268
可执行语句，用语句块分组，83
可重用性，268
客观的，19~21
客观系统，559~560
客户代码，21
 保护堆数据对象，406
 进行内存管理，333~334
 使用返回的对象，336~340
控制流，83~128，772
 表达式，83~84
 条件语句，84~104
 标准形式，85~88
 常见错误，88~98
 嵌套条件，98~104
 循环，104~117
 语句，83~84
 转移语句，117~127
 break语句，117~120
 continue语句，120
 exit语句，121~125
 goto语句，120~121
 return语句，121~125
 switch语句，125~127
块注释，27~30
块作用域，167，168，172
快速原型，8~9

L

类，278
 抽象，781~782
 存储，175~186
 非数值类的运算符重载，400~414
 复合，439~482
 嵌套，478~480
 嵌套模板类，719~722
 容器，465~482
 相似，484~340
 友元，480~482
类Name，327，328
类成员
 控制对……的访问，314~319
 重载运算符函数，367
类对象，异常，748~757
类复合，779~781
 用继承代替，513，695~696，700，722
类构造，相似类，485
类继承，779~781
类聚集，441
 C++语法，442~443
类可见性
 继承，584~585
 将任务推向服务器类，582~583
 类之间的关系，580~582
 任务划分，580~585
类设计重用，简单例子，694~701
类协作，440
类型，31~33
类型定义，37
类型名，168
类型修饰符，769
类型转换，249~250
类型转换运算符，757~767
 const_cast运算符，761~763
 dynamic_cast运算符，764~766
 reinterpret_cast运算符，761
 static_cast运算符，757~761
 typeid运算符，766~767
类作用域，167
连续数组，131，168
处理的数据存入连续数组，147
联合，157~160，166，774
 标志域，159~160
 定义，157，774
链表达式，430~433
流程控制构造，38~47
 函数调用，41~47
 条件语句，37~38
 循环，38~40
逻辑运算符，73~75，771
逻辑运算符，69

M

枚举，160~162，166，774
 定义，157
美国国家标准协会(ANSI)，19
面向对象方法，1~21
 程序组件的复杂性，10~11
 对象，16~17
 定义，16
 使用的优点，17~18
 数据，16~17

风险和回报，10
软件危机的起因，2~5
设计目标，10~12
设计问题，15~16
对象初始化，15~16
命名冲突，15
设计质量，11~15
将数据与函数绑定在一起，12~14
内聚，11
耦合，11~12
信息隐藏与封装，14~15
面向对象软件工程，559~560
名字空间作用域，167
名字作用域，167~175
命名冲突，15
模板，782
模板，694~729，782
静态成员，722~723
类设计重用的简单例子，694~701
模板规则说明，724~728
模板函数，728~729
实现，704~710
模板类定义的语法，701~711
模板类实例之间的关系，716~723
模板类说明，702~703
模板实例化，703~704
嵌套模板，710~711
嵌套模板类，719~722
有多个参数的模板类，61~66
常量表达式参数，713~716
类型参数，711，713
作为友元的模板类，717~719
模板规则说明，724~728
例子，725~726
模板类实例，716~723
目标，769~770

N

内部数据类型，769~771
内存管理，166~223
磁盘文件的输入和输出，215~223
从文件读入，217~221
输出到文件，215~217
输入/输出文件对象，221~223
存储类别，175~186
定义，174
静态变量，182~186
外部变量，178~182
自动变量，176~178
动态的，186，197~198
动态结构，207~215
动态数组，197~207
堆，166，186~215
分配内存，191~194
名字作用域，167~175
C++词法作用域，167
数组和指针，194~196
运算符和函数调用，333~335
栈，166
作为类型变量的C++指针，187~191
内存泄漏，428~429
内聚性，270
设计质量，11
内联函数，255~257

O~P

耦合度，271~281，771~773
降低强度，276~281
设计质量，11~12
显式的，274~277
隐式的，271~274
派生，495
派生类，508，516~517，519~523
access() 方法，600~603，604~606
传递指针与引用参数，609~610
使用指针访问对象，603~604
派生类指针，指向基类对象，634
抛出类对象，例子，751~752
抛出异常，736，737~738
例子，740
瀑布方法，7~9

Q

前缀加1运算符，例子，667~668
前缀运算符，68~69，771
嵌套类，478~480
嵌套模板，710~711
嵌套模板类，719~722
嵌套条件，98~104

嵌套循环和多维数组，137
嵌套作用域
 运算符，174
 在……中使用相同名字，171~174
强类型，589，591，612
 转换构造函数，670
全局变量，178~182
 extern关键字，179~182
 从其他文件引用，178~180
 进行内存分配，178~179
 静态的，184~186
 使用的优点，178~179
 作为静态变量定义，183
 作为类特性使用，345~347
 作用域，172
缺省，支持，770
缺省继承模式，518~520

R

人的智能
 重用，545~547
 例子，546
容器类，465~482
 动态分配内存，477~478
 嵌套类，478~480
 友元类，480~482
 有固定大小的组件和数组溢出，468~469
 迭代器，475~476
 溢出控制，473~474
冗长，9
软件设计目标，10~12
软件危机
 程序员，摆脱，5~7
 改进管理技术，7~9
 快速原型，8~9
 瀑布方法，7~9
 起因，2~5
 设计复杂而完善的语言，9
软件系统
 把函数分成独立的操作，3~4
 测试，4
 规格说明，3
 维护，3
 系统任务，复杂性，4
 系统组件，互依赖性，3~4
弱类型，591

S

设计，769~770
设计复杂而完善的语言，9
设计目标，10~12
设计问题，15~16
 对象初始化，15~16
 命名冲突，15
设计质量，11~15
 将数据与函数绑定在一起，12~14
 内聚，11
 耦合，11~12
 信息隐藏与封装，14~15
声明，30，33~35
 定义，33
声明异常，737，743~745
十六进制数，57
实例化
 模板，703~704，716~723
 类之间的关系，716~723
实现阶段，瀑布方法，7
使用标准转换对指针进行转换，例子，759~760
使用继承关系的异常类，例子，753~754
使用模板类的优点，702~703
使用全局变量对对象实例计数，346
受保护继承，511~515
 访问派生类对象的基类成员，512~513
 重用代码例子，553~554
输入/输出运算符，687~693
 重载运算符<<，691~693
 重载运算符>>，688~691
输入数据溢出，143~144
数据成员
 常量数据成员，458~459
 方法参数，访问，445~446
 访问类数据成员，443~445
 静态，作为其类自身的成员，463~465
 具有特殊属性的，457~465
 引用数据成员，459~461
 用类对象作为数据成员，440~446
数据封装，281~286，291~301，778
数据和表达式，55~82
数组，166，773~774
 C++数组，定义，131~133
 操作，133
 常见的局限，208
 初始化，132，133

传递，246~249
大小，130~131
定义，131
动态的，197~207
多维，136~138
C++支持，138
初始化，136
存取多维数组，137
定义其大小，136
嵌套循环，137
防止溢出，输入数据，148~149
岗哨值，131
连续的，131
联合，157~160
定义，157
枚举，160~162
定义，157
数组类型，定义，150~152
数组溢出，141~144
防止，143
在插入算法中，146~150
在串接中，143~144
同种类聚集，129~151
位域，162~164
定义，157
稀疏的，130
下标有效性检查，133~136
严格的限制，130
一个元素的位置，130
元素，153
指针，194~196
字符，138~144
二维，144~146
字符函数和内存此用，141~144
作为值向量，129~131
数组变量，132
数组类，674~683，702~703
数组模板类，724~728
数组声明，132
数组数据
使用同一成员函数，680~681
熟悉，677
使用重载下标运算符，682~683
数组溢出，674~676
无效的下标值，674~676
作为整数成员的容器，677~678
数组溢出，141~144
防止，144，148
可以防止数组溢出的事务数据输入程序，197~198
在插入算法中，146~150
在串接中，143~144
防止，144，148
私有继承，515~517
私有原型，将对对象的不正确处理视为非法的，437~438
算术赋值，772
算术运算符，67~69

T

体系结构设计阶段，瀑布方法，7
条件语句，37~38
标准形式，85~88
常见错误，88~98
嵌套条件，98~104
条件运算符，76~77，772
通过公共继承并重新定义方法的重用的例子，555~556
通过继承相关的类之间的转换，594~612
安全转换与不安全转换，595~599
对象的指针与引用的转换，599~607
指针与引用参数的转换，607~612
同构聚集
作为结构，151~152，157，161，318~319，616
作为数组，129~151
统一建模语言，499，541，557~565
标识符
多重性的表示，563~565
关系的表示，561~562
聚集和泛化的表示，562~563
类的表示，560~561
使用的目的，557~560
图形用户接口（GUI），783

W

外部变量，178~182
外部声明，与其他文件交流，180~181
微软，2
为了客户代码使用访问函数的例子，305
维护阶段，瀑布方法，7
make()，683~684
malloc()，193，321，326，333~335
内存管理，193~194

未初始化的指针，189
位移，771
位域，157，162~164，166
 定义，157
谓词，442
文件作用域，167、168，169
无符号字符，775
无条件跳转，772
无约束类型，719

X

析构函数，326~330，778~779
 调用时间，330
 多继承，644~645
 函数，326
 继承中，537~539
 虚函数，639~640
 语法，326
稀疏数组，130
系统分析阶段，瀑布方法，7
系统支持的赋值运算符，问题，427~428
下标有效性，133~136
下标有效性检查，133~136
下标运算符，674~683
显式耦合，274~277
相等运算符，72~73，771
相似操作，588
相似对象，588
相似类，484~340
 C++继承
 调整对Derived类中的成员的访问权限，517~518
 访问基类和派生类服务，503~506
 访问派生类对象的基类成员，506~520
 公共继承，507~511
 名字重载与名字隐藏，522~525
 派生类所隐藏的基类方法的调用，525~529
 缺省继承方式，518~520
 使用进程改进程序，529~532
 受保护继承，511~515
 私有继承，515~517
 用又链接相关类，495~497
 语法，497~503
 作用域规则和名字解析，520~522
 程序完整性，保持任务推向服务端，488~492
 处理，485~497
 继承中的析构函数，537~539
 类构造，485
 派生类的构造函数，532~545
 为服务器对象建立单独的类，492~494
 在类中合并不同功能，例子，487~488
详细设计阶段，瀑布方法，7
消费者保护法，2
信息隐藏，14~15
 主要优点，14
星号指针标识作用域，187
行结束注释，27~30
修饰符，442~443
虚函数，612~640，612~615
 virtual关键字，629~636
 纯，636~639
 定义，781
 动态绑定，633~636
 传统方法，615~622
 面向对象的方法，622~629
 静态绑定，633~636
 析构函数，639~640
需求定义过程，瀑布方法，7
选择符，442
循环，38~40
 do-while循环，104，111~114，772
 do循环，20
 for循环，20，114~117
 while循环，20，105~111
 带格式化输出，例如，39
 嵌套的多维数组，137
循环变量的作用域，174~175

Y

验收测试阶段，瀑布方法，7
依赖性，316
移位运算符，69~70
移位运算符，69
以重新定义函数的方式继承，554~556
异常，782~783
 按位逻辑运算符，70~72
 表达式，34~40
 逗号运算符，77
 赋值运算符，75~76
 高优先级运算符，66
 关系运算符，72~73
 逻辑运算符，73~75

算术运算符，67~69
条件运算符，76~77
相等运算符，72~73
移位运算符，69~70
运算符总结，66
异常，定义，730
异常处理，730~768，782~783
C++异常的语法，736~748
简单例子，730~736
类对象异常，748~757
标准异常库，756~757
抛出/声明/捕获异常的语法，749~752
使用异常的继承关系，752~756
类型转换运算符，757~767
const_cast运算符，81，761~764
dynamic_cast运算符，81，764~766
reinterpret_cast运算符，81，761
static_cast运算符，81，757~761
typeid运算符，766~767
异常处理机制，736
定义，736
异常的机制，730
异常的语法，736~748
捕获异常，736，738~743
错误相关的源代码，组织，736~737
抛出异常，736，737~738
声明异常，737，743~745
异常处理机制，736
异常处理机制，操作，736
重新抛出异常，745~748
异常类，736，756~757
异构列表处理
传统方法，620~622
面向对象方法，627~629
使用虚函数，631~633
引用变量，238~241
引用数据成员，459~461
隐式耦合度，271~274
映射，719
用函数封装，281~286，291~301
缺点，299~301
用函数进行面向对象程序设计，268~302
封装，281~286，291~301
缺点，299~301
内聚性，270
耦合，271~281
降低强度，276~281
显式的，274~277
隐式的，271~274
信息隐藏，286~291
友元函数，383~394
友元类，480~482
有符号的数值，57
有值参数的重载的串接函数，412~413
右值，75
与C语言相比，17~21
语句，34~40，83~84
程序开发工具，51~54
定义，30~33
复合的，36~38
赋值，37
函数调用，37
类，47~51
类型定义，37
流程控制结构，38~47
函数调用，41~47
条件语句，37~38
循环，38~40
条件的，84~104
同一行，执行，38
语句块，36~38
语句块
将语句合并，84
嵌套，772
预处理程序指令，24~27
原子变量，129
源代码，函数，12
运算符::，174
运算符函数，353~394
复合对象，353
混合参数类型，377~383
友元函数，383~394
有理数案例分析，370~377
运算符重载，353~360
限制，360~364
作为类成员，364~369
重载的例子，358
运算符重载，69，353~360，651~693
二元运算符，651，659~674
后缀重载运算符，666~668
减量运算符，659~666
增量运算符，659~666
转换运算符，668~674
非数值类，400~414

String类，401~402
保护程序完整性，409~413
保护堆数据对象，406~407
堆内存的动态管理，402~406
防止内存泄漏，408
重载的串接运算符，406~407
函数调用运算符，683~687
简介，651~658
输入/输出运算符，687~693
重载运算符<<，691~693
重载运算符>>，688~691
下标运算符，674~683
限制，360~364
不可重载的运算符，360~361
参数个数，363~364
返回类型，361~362
运算符优先级，364
作为类成员重运算符，364~369
Const关键字，368~369
用类成员取代全局函数，364~366
在链式操作中使用类成员，366~367
运行时检查，134

Z

在独立的作用域中使用相同的名字，170~171
在客户端代码中使用返回的对象，336~340
返回对象，338~340
返回指针和引用，336~338
在类中合并不同功能，例子，487~488
早期绑定，613~615
栈，166，694~695
整数类型，57~64
布尔值，64
修饰符，59~62
字符，62~64
操纵字符数值，63
例子，63
整数溢出的例子，58
整数字面值，61
整型，57~64
布尔值，64
修饰符，59~62
字符，62~64
整型值，32
直接超类，496
值，55~57
按值调用函数，230~231
值语义，419~420
指向，187
指向派生类对象的基类指针，634
指针，166
操作，189
处理数组，193
对无名堆变量使用，191~1+2
对一般命名变量使用，189
通过指针调用函数，232~238
指针变量，187~191，232
指针悬垂，253
注释，27~30
转换构造函数，325~326
例子，673~674
转换运算符，668~674
转移语句，117~127，129
break语句，117~120
continue语句，120
exit语句，121~125
goto语句，120~121
return语句，121~125
switch语句，125~127
转义字符，62~63
子类型，495
字符，775
字符，对待，770
字符串函数，内存说用，141~144
字符类型，62~64
操纵字符数值的一个例子，63
字符数组，138~144
0终止符，139
操作，140~141
定义，138~140
二维，144~146
字符字母，770
字面值，62
自动变量，176~178
函数的形式参数，178
名字，177
自我赋值，429~430
组件的层次结构，153~154
作为成员函数的构造函数，319~321
作为传统程序设计语言，769~772
作为面向对象语言，778~783
作为模块化语言，772~778
作为向量的数组，129~131

作业控制语言(JCL)，132
作用域，21
C++词法作用域，167
定义，166、167
独立的作用域，在其中使用相同名字，170~171
对象实例，331~333
类型名，32
嵌套的，171~174
同一作用域的名字冲突，167~170
循环变量作用域，174~175

作业控制语言(JCL)，132
作用域，21
C++词法作用域，167
定义，166、167
独立的作用域，在其中使用相同名字，170~171
对象实例，331~333
类型名，32
嵌套的，171~174
同一作用域的名字冲突，167~170
循环变量作用域，174~175